Software Quality Journal (2023) 31:917–946
https://doi.org/10.1007/s11219-023-09615-7

An empirical evaluation of defect prediction approaches in within-project and cross-project context

Nayeem Ahmad Bhat[1] · **Sheikh Umar Farooq**[1]

Accepted: 14 January 2023 / Published online: 4 March 2023

Abstract
The software defect prediction approaches are evaluated, in within-project context only, with only a few other approaches, according to distinct scenarios and performance indicators. So, we conduct various experiments to evaluate well-known defect prediction approaches using different performance indicators. The evaluations are performed in the scenario of ranking the entities — with and without considering the effort to review the entities and classifying entities in within-project as well as cross-project contexts. The effect of imbalanced datasets on the ranking of the approaches is also evaluated. Our results indicate that in within-project as well as cross-project context, process metrics, the churn of source code, and entropy of source code perform significantly better under the context of classification and ranking — with and without effort consideration. The previous defect metrics and other single metric approaches (like lines of code) perform worst. The ranking of the approaches is not changed by imbalanced datasets. We suggest using the process metrics, the churn of source code, and entropy of source code metrics as predictors in future defect prediction studies and taking care while using the single metric approaches as predictors. Moreover, different evaluation scenarios generate different ordering of approaches in within-project and cross-project contexts. Therefore, we conclude that each problem context has distinct characteristics, and conclusions of within-project studies should not be generalized to cross-project context and vice versa.

Keywords Cross-project defect prediction · Software quality assurance · Source code metrics · Process metrics · Churn of source code · Feature selection · Imbalance learning

1 Introduction

Defect prediction, one of the holy grails of software development, has attracted much attention among software engineering researchers. The driving factor is resource allocation. Quality assurance resources being limited, it is wise to prioritize and allocate the

✉ Nayeem Ahmad Bhat
bhat.nayeem2@gmail.com

Sheikh Umar Farooq
suf.cs@uok.edu.in

[1] Department of Computer Sciences, North Campus, University of Kashmir, Srinagar, J&K, India

resources towards the areas with higher probable defect scores. Different approaches relying on various information sources, like as code metrics Nagappan and Ball (2005), Zimmermann et al. (2007), process metrics Rahman and Devanbu (2013), Mnkandla and Mpofu (2016), previous defects Kim et al. (2007), Felix and Lee (2017), the churn of source code, the entropy of source code D'Ambros et al. (2012), and entropy of changes Hassan (2009) have been devised to perform the task of defect prediction. The relative performance comparison of these approaches is still one of the areas where research is needed. Most of these approaches were compared to only a few other approaches, or were evaluated using distinct performance indicators, or were compared only in the within-project context. Moreover, replicating the evaluations is also a difficult task since the data of commercial systems not available for public use was used in the evaluations.

An evaluation of the relative performance of different approaches in the cross-project context is imperative to identify the approaches with stable and good performance. Traditionally, most defect prediction models were trained and evaluated in the within-project context, i.e., the data of past releases of the software were used for model training to predict defects in upcoming releases. But, there were software systems for which no or scarce defect data was tracked in defect tracking systems. For example, no defect data is available for the first release of a software. Moreover, people prefer the reuse of old defect data in defect prediction Turhan et al. (2009). The local data scarcity and the possibility of old data reuse triggered the idea of cross-project defect prediction (CPDP). In CPDP, defect data of multiple source projects combined together is used in training a prediction model that is used to predict defects in a target project. In the past decade research focused on designing training data selection Bhat and Farooq (2021a, b) and Li et al. (2017) and transfer learning Xu et al. (2019), Qiu et al. (2019), Ma et al. (2012) techniques for alleviating the distribution mismatch between different projects. The relative performance of different data sources (features) used in model training has not been evaluated in the cross-project context. Moreover, the defect datasets are implicitly imbalanced because only a small proportion of modules/classes contain most of the defects. The use of these imbalanced datasets biases the trained models towards correctly classifying the majority class non-defective instances and incorrectly classifying the minority class defective instances. Various techniques have been proposed and evaluated to address the effect of imbalanced datasets on the prediction results. However, the effect of class imbalance on the ranking of different approaches has not been evaluated.

Performance evaluation of the approaches is also addressed differently. Some researchers use classification (i.e., predicting if an artifact is defective or non-defective), while others use a ranking of artifacts with or without taking effort factor (i.e., the effort required to inspect an artifact) into consideration.

Therefore, we provide a performance evaluation of different approaches over three different scenarios of classification, ranking and ranking with effort consideration in within-project as well as cross-project defect prediction contexts. The evaluation is performed over five projects from publicly available Bug Prediction Dataset. The evaluation is limited at comparing different data sources for prediction performance. The effect of balanced datasets on the ranking of different data sources is also evaluated. The evaluations do not compare the learning algorithms, or data preprocessing methods.

The primary contributions of this paper are:

1. An evaluation of different defect prediction approaches in the within-project context.
2. An evaluation of different defect prediction approaches in the cross-project context.

3. An evaluation of the effect of data balancing on the ranking of different defect prediction approaches in both within-project and cross-project context in classification scenario.

The evaluations are performed according to three different scenarios of classification, ranking, and ranking with effort consideration in two distinct ways, aimed at (1) comparing performance and (2) testing the statistical significance of performance differences.

The remainder of this paper is structured as follows: First, we give a summary of the related work in Section 2. Next, we summarize the datasets and data sources used in our evaluations in Section 3. Afterward, in Section 4, we discuss the evaluation scenarios we used for the evaluation of approaches. We discuss the experimental methodology used to evaluate the approaches in the within-project and cross-project contexts in Sections 5 and 6, respectively. In Section 7, we report and discuss the within-project and cross-project defect prediction experimental results. In Section 8 we report on the statistical significance of experimental results. In Section 9, we discuss experimental methodology for evaluating the effect of class imbalance on the ranking of different approaches and discuss the results. In Section 10, we discuss the threats to the validity of our work. Finally, in Section 11 we conclude.

2 Related work

In defect prediction, we train statistical or machine learning models to predict the defect proneness in software components. The predicted defect proneness score is used to prioritize the code review and testing effort optimally Suhag et al. (2020). Hosseini et al. in a systematic literature review Hosseini et al. (2019) report there exists a lot of diversity in the features, datasets, performance evaluation, and training methods used in software defect prediction. They further report that CPDP is a challenge that needs more scrutiny before it is used reliably in practice. In this section, we summarize various defect prediction approaches, the type of data they need, and the datasets they were validated on.

Change log approaches use information collected from the versioning system, supposing that recently or frequently changed artifacts are the greatest possible source of future defects. Khoshgoftaar et al. (1996), used the number of past changes to the software modules, to predict the modules as defect prone. At the module level, they declared that the number of lines of code added or removed in the past predicts future defects with good performance.

Nagappan and Ball in Windows Server 2003 study the impact of code churn (i.e., the number of changes) on the defect density. They conclude that relative churn performed better than absolute churn metrics Nagappan and Ball (2005).

Moser et al. (2008) used different metrics (code churn, past defects, and refactorings, number of authors, age and size of files, etc.) to classify the files of Eclipse as defective or non-defective.

Hassan (2009) proposed the entropy of change metrics (i.e., the complexity of code changes). They compared the entropy to code churn, previous defects, and found the entropy to perform better. Their evaluation used six open-source systems: FreeBSD, KDE, KOffice, NetBSD, OpenBSD, and PostgreSQL.

Hassan and Holt (2005) validate the heuristics regarding the defect proneness of the most recent changes and most number of bug fixes in the files on six open-source systems: FreeBSD, KDE, KOffice, NetBSD, OpenBSD, and PostgreSQL. They conclude that most recently changed and fixed files are most defect prone.

Kim et al. (2007) similar to Hassan and Holt (2005) use the features of recent changes and defects with an additional assumption that defects occur in bursts.

Rahman and Devanbu (2013) across 85 releases of 12 large open-source projects, built prediction models to check the performance, portability, stability, and stasis of process and product metrics. Their findings indicate that process metrics are generally more useful than widely used product metrics for prediction.

Single version approaches with a diverse set of metrics analyze the present state of the source code rigorously, presuming that the current design and behavior of software determines the presence or absence of future defects more than the history of software. One standard set of metrics is the set of source code metrics proposed by Chidamber and Kemerer (1994). Ohlsson and Alberg (1996) used some graph metrics entailing, cyclomatic complexity, to predict defects in telecom switches. Nagappan et al. (2006) on five Microsoft sytems used a list of source code metrics to predict release-level defects. Although the predictors were able to predict well for individual projects but failed to do so on all projects.

D'Ambros et al. (2010, 2012) evaluate the performance of various source code metrics (CK, OO) along with change log approaches (process metrics, previous defects, entropy of changes, churn of source code, entropy of source code).

Cross-project defect prediction Traditionally, the historical data of prior releases of software is used to train defect prediction models for predicting defects in the future releases of the software — an idea called within-project defect prediction. However, there are software systems for which insufficient defect data is recorded in defect tracking systems. For example, no defect data record exists for the first release of a software system Çatal (2016). Moreover, the reuse of old data in defect prediction is preferred over the collection of new data for each project Turhan et al. (2009). The insufficiency of local defect data and the potential for reuse of defect data of other projects gave birth to the idea of cross-project defect prediction (CPDP). In CPDP, for software systems with insufficient training data, models trained on defect data of other software are adapted for the defect prediction task.

Zimmermann et al. (2009) analyzed the impact of the domain, process, and code structure on CPDP. Only 3.4% among 622 cross-project defect predictions over 12 real-world applications, satisfy their performance benchmark, and the CPDP models trained on one set of software did not generalize to other software. Turhan et al. (2009) report an increased defect detection rate with an associated increase in false-positive rate in defect predictors that are trained on all multi-source cross-project data. They introduced an analogy-based training data selection method (Burak filter) to select the relevant training data and reduced the false-positive rate. Bhat and Farooq (2021) propose a filter approach (BurakMHD filter) for selecting relevant training data in CPDP and conclude the BurakMHD filter compared to Burak filter Turhan et al. (2009) and Peter filter Peters (2013) improves the CPDP performance. Turhan (2012) asserted the dataset shift problem between software defect datasets is the main reason for the substandard performance of CPDP.

Hosseini et al. (2018) infer that search-based methods integrated with feature selection are a propitious way for training data selection in CPDP. Yu et al. (2019) through an empirical

study on NASA and PROMISE datasets reveal that feature subset as well as feature ranking approaches improve the CPDP performance. Therefore they recommend selecting the representative subset of features to improve the CPDP performance. Xu et al. (2019) present a Balanced Distribution Adaptation Based Transfer Learning technique for CPDP and report their approach performs better than 12 baseline approaches. Sun et al. (2021) introduced the Collaborative filtering-based source projects selection (CFPS) technique for source project selection and validated the feasibility, importance, and effectiveness of source projects selection for CPDP. Agrawal and Malhotra (2019) report, the CPDP is not always feasible. They suggest the selection of relevant training for the defect prediction task.

Class imbalance learning The defect datasets are implicitly imbalanced because only a small proportion of all the modules/classes contain most of the defects and most of the classes/modules are without any defects Wang and Yao (2013). The imbalanced nature of the datasets biases the classification models towards correctly classifying the non-defective class instances and classifying the defective class instances incorrectly Haixiang et al. (2017). To improve the performance of models trained on imbalanced datasets both data-level approaches García et al. (2012), Chawla et al. (2002), Barua et al. (2014), Al Majzoub et al. (2020), Han et al. (2005), Menardi and Torelli (2012), Bashir et al. (2020), Bennin et al. (2022), Feng et al. (2021), Malhotra and Jain (2022), Feng et al. (2021), Bennin et al. (2017) that alter the distribution of the datasets and algorithm-level approaches Zhou and Liu (Jan 2006), Tomar and Agarwal (2015, 2016), Ryu et al. (2017) that modify the learning procedure according to the costs function have been proposed Dar and Farooq (2022). Bennin et al. (2017) studied the effect of resampling approaches on the classification performance. They observed recall and G-mean are improved at the expense of PF and no significant effect on AUC indicating that the resampling approaches improve the defect classification but not defect ranking or prioritization. Limsettho et al. (2018) introduced the CDE-SMOTE technique to cope with the class imbalance and distribution difference between source and target projects. Goel et al. (2021) evaluate data sampling techniques used to cope with the class imbalance in CPDP. And conclude, the synthetic minority oversampling technique (SMOTE) is suitable to handle the class imbalance. Han et al. on the assumption that the borderline instances are most likely to be misclassified Han et al. (2005) proposed the BLSMOTE technique that synthetically creates minority instances from the borderline minority instances.

Effort-aware defect prediction Traditional methods largely ignore the effort required to inspect an artifact, during defect prediction. They assume that the effort is uniformly distributed across modules. Arisholm et al. (2010) propose the effort is approximately proportional to the size of the software modules. Hence, to discover an equal amount of defects, less effort is required in shorter files. The idea of effort-aware defect prediction is to output a ranking of files in which a lesser amount of effort would discover a greater number of defects.

Mende and Koschke (2009) showed that when evaluated with traditional evaluation metrics like the ROC curve, the simplest defect prediction model — *large files are the most defect prone* — performs well. However, under effort-aware evaluation, the simplest model performs the worst. Similarly, when Kamei et al. (2010) introduced effort-aware performance metrics to revisit the common findings in defect prediction, they confirmed the fact that process metrics still outperform the product metrics.

We observe that the evaluation of the relative performance of different data sources for defect prediction is still one of the areas where research is needed. While most of the earlier studies used different data sources, the data sources were compared to only a few other data sources, or were evaluated using distinct performance indicators without any effort-aware consideration, or were compared only in the within-project context. The effect of data imbalance on the relative performance of different data sources was also not considered. Moreover, it is also difficult to replicate those studies since data of commercial software not accessible for public use was used. Therefore, it is imperative to evaluate the data sources in both within-project and cross-project contexts using different evaluation scenarios (classification, ranking, and ranking with effort consideration). It is also important to evaluate the effect of data imbalance on the ranking of different data sources.

3 Evaluated data sources

It is impractical to compare the plethora of existing approaches in defect prediction. To have a range as inclusive as possible for evaluation, we select the approaches summarized in Table 1. We select five project datasets summarized in Table 2 from the Bug Prediction Dataset D'Ambros et al. (2010) publicly accessible at https://bug.inf.usi.ch/index.php. Only Bug Prediction Dataset is used in the experiments because no other dataset provides information about all the approaches summarized in Table 1. Source code metrics — CK metrics Basili et al. (1996); Capretz and Xu (2008) and an additional set of OO metrics D'Ambros et al. (2010), previous defect data Zimmermann et al. (2007); Kim et al. (2007), change metrics Moser et al. (2008), and entropy of changes Hassan (2009), the churn of source code, and entropy of source code D'Ambros et al. (2010) of the five projects from Bug Prediction Dataset are used in evaluations. To help replication of our experiments we select the publicly accessible dataset for defect prediction. Moreover, the five projects have the same Java code structure and the datasets have been recorded by the same defect tracking methods.

4 Performance evaluation

The evaluation of defect prediction is still a matter of debate Zhang and Zhang (2007), Menzies et al. (2007), Lessmann et al. (2008), Jiang et al. (2008). In line with a particular usage scenario of defect prediction, various strategies are used to make performance evaluations of defect prediction approaches. We use the performance metrics commended by Jiang et al. (2008) to evaluate the approaches in the scenario of classification (defective or non-defective), ranking (most defective to least defective), and effort-aware ranking (most defect dense to least defect dense).

Classification One common setting in which defect prediction is applied is the classification wherein we are focused on the grouping of the classes in defective and non-defective groups. Moreover, the underlying classification models are often probabilistic classifiers that assign probabilities instead of class labels to each observation Fawcett (2006). They are converted to binary classifiers by a user-defined threshold. And the commonly used evaluation

Table 1 Evaluated Defect Prediction Approaches

Category	Predictor	Description	Used by
Process metrics	MOSER	Set of change metrics	Moser et al. (2008)
	NFIX-ONLY	Number of bug fixes in the past	Zimmermann et al. (2007)
	NR	Number of revisions	Graves et al. (2000)
	NFIX+NR	Combined NFIX and NR metrics	D'Ambros et al. (2010)
Previous defects	BUG-CAT	Number of actual bug fixes in the past categorized according to severity and priority	D'Ambros et al. (2010)
	BUG-FIX	Number of actual bug fixes in the past	Kim et al. (2007)
Source code metrics	CK+OO	Chidamber and Kemerer suite plus an additional set of object oriented metrics	D'Ambros et al. (2010)
	CK	Chidamber and Kemerer suite of metrics	Basili et al. (1996)
	OO	Set of object oriented metrics	D'Ambros et al. (2010)
	LOC	Lines of Code	Gyimothy et al. (2005)
Entropy of changes	HCM	History of Complexity Metric	Hassan (2009)
	WHCM	Weighted History of Complexity Metric	
	EDHCM	Exponentially Decayed History of Complexity Metric	
	LDHCM	Linearly Decayed History of Complexity Metric	D'Ambros et al. D'Ambros et al. (2010)
	LGDHCM	Logarithmically Decayed History of Complexity Metric	
Churn of source code metrics	CHU	Churn	D'Ambros et al. D'Ambros et al. (2010)
	WCHU	Weighted Churn	
	LDCHU	Linearly Decayed Churn	
	EDCHU	Exponentially Decayed Churn	
	LGDCHU	Logarithmically Decayed Churn	
Entropy of source code metrics	HH	Entropy	D'Ambros et al. D'Ambros et al. (2010)
	HWH	Weighted Entropy	
	LDHH	Linearly Decayed Entropy	
	EDHH	Exponentially Decayed Entropy	
	LGDHH	Logarithmically Decayed Entropy	

measures of precision and recall assess the performance according to a particular threshold. As a surrogate metric, we use the Receiver Operating Characteristic (ROC) curve. ROC curve is a way to measure the performance of the prediction model in general, as we sweep over the range of different thresholds. The ROC curve plotted between sensitivity and specificity illustrates the benefits of using the model (true-positives) versus the costs of using the model (false-positives) at different thresholds. The area under the ROC curve (AUC), a scalar value, $0.5 <= AUC <= 1.0$, is the estimated probability that a randomly picked positive instance will be set with a higher predicted p by the prediction model than another randomly picked negative instance Hanley and McNeil (1982). Hence, the statistic measures a model's capability to correctly rank instances. A perfect model has an AUC of 1, while the worst model has an AUC of 0.5. To perform a comprehensive comparison across approaches over different datasets we report the AUC. AUC is appropriate to compare classifiers over different datasets Lessmann et al. (2008) and is often used for that purpose.

Ranking A more realistic and useful scenario of defect prediction is that where the modules are ranked on their defect proneness score, and the ranking is used to prioritize the quality assurance effort (testing and code inspection). The overall concept is known as Module-Order-Model (MOM) Khoshgoftaar and Allen (2003). Consequently, the evaluation of how good a model is at ranking instances correctly is required. One way to graphically evaluate Module-order-Models is lift charts, also known as Arberg diagrams Ohlsson and Alberg (1996). They are generated by ordering the modules according to the defect score set by a prediction model and illustrating for each ratio of modules on the x-axis which ratio of defects has been predicted on the y-axis. For instance, Ostrand et al. found 83% of the defects in 20% of the files Ostrand et al. (2005). Mende and Koschke have defined a comprehensive performance evaluation measure p_{opt} from lift charts, by comparing a prediction model with an optimal model Mende et al. (2009). An optimal model is generated by ordering all the modules by decreasing defect score. Δp_{opt} is interpreted as the difference between the area of the lift curve of the optimal model and the area of the lift curve of the prediction model. A higher value of Δp_{opt} indicates a greater difference between the optimal model and the prediction model, and hence a bad prediction model. They define $p_{opt} = 1 - \Delta p_{opt}$, where a high value means a better model.

Effort-aware ranking Arisholm et al. argue that the costs of testing or reviewing of software modules are not uniformly distributed but relate to the size of software modules Arisholm et al. (2007) to some extent. They introduce a variant of lift charts where the x-axis contains the ratio of lines of code instead of ratio of modules. To make an effort-aware evaluation, we define a measure Relative defect risk factor of software modules as $RDR(x) = errors(x)/effort(x)$. Similar to Mende and Koschke (2010), Menzies et al. (2010), and D'Ambros et al. (2012), we use the LOC metric as a notion of effort and order the software modules according to the decreasing defect density or RDR. The idea is that larger modules take much more time to review than smaller modules. Therefore, if the number of predicted defects is the same then prioritize the smaller modules before larger modules. Accordingly, we use LOC-based cumulative lift charts to evaluate the performance of the prediction approaches. LOC-based cumulative lift charts are generated by ordering the modules according to their defect density and illustrating for each ratio of lines of code on the x-axis which ratio of defects has been predicted on the y-axis. Similar to the ranking without effort consideration scenario, we again use the p_{opt} to evaluate the performance of the approaches. But in the present case, to distinguish it from the previous metric p_{opt} we refer to it as p_{effort}.

Table 2 Five Bug Prediction Datasets used in evaluations, sorted in order of number of Lines of Code

Project	LOC	(# classes)	post-release-defects	% defects
Eclipse	224055	997	374	20.66
Mylyn	156102	1862	340	13.16
Pde	146952	1497	341	13.96
Lucene	73184	691	97	9.26
Equinox	39534	324	244	39.81

The two desirable properties of this performance measure are: It takes costs related with reviewing or testing the code into consideration, and it considers the actual distribution of defects by comparing against the theoretically possible optimal model.

5 Experiment 1: within-project defect prediction context

The 25 defect prediction approaches listed in Predictor column of Table 1 are used in each software system to train and validate the models. For within-project comparison of approaches, the methodology discussed below and given in Algorithm 1 is followed during model training and validation for the 25 defect prediction approaches.

Algorithm 1: Outline of the within-project defect-prediction

```
DATA = [Eclipse, Mylyn, Equinox, PDE, Lucene]
foreach data in DATA do
    // Let P be the set of predictor metrics and brow the
       number of post release bugs in data
    (P, b) = apply min-max normalization(P, brow)
    // if classification scenario is pursued, b is a
       boolean vector(buggy or clean)
    if classification then
     | b = b > 0
    end
    // select the best features to fit b doing 10-fold
       cross validation
    Pbest = wrapper(P, b, 10-folds, "glm")
    folds = stratified partition(b, 5-fold, 10-repeats)
    performance = crossvalidation(Pbest, b, folds, "glm")
end
```

5.1 Min-Max normalization

Predictors measured at different scales contribute differently to the model fitting process and might lead to a bias in the process. To handle the potential bias feature normalization such as Min-Max scaling is applied before fitting a model Henderi et al. (2021), Patro and Sahu (2015), Jain and Bhandare (2011).

5.2 Feature selection

A multitude of features might result in model overfitting which drastically degrades the model predictive performance when applied to new data. The multicollinearity among features is problematic because it is difficult to interpret the effect of individual features. A feature selection technique called sequential floating forward selection (SFFS) Pudil et al. (1994) is used to select the best subset of features that give the best predictive performance. The SFFS selects a subset of features by sequentially adding and deleting features to the subset as long as the performance increases.

To start with an empty subset of features, for each unselected feature along with the selected features, the wrapper performs a stratified tenfold cross-validation experiment and adds the feature that improves the performance to the subset in a sequential manner. After adding a feature to the subset, the wrapper checks for the worst feature, elimination of which might improve the performance and eliminate that from the subset. The addition and elimination are repeated until adding more features does improve the performance and finally stopped when no improvement is seen in the performance.

5.3 Training regression models

We train generalized linear regression models from the features we compare in our experiments. The explanatory variables — or the features used for prediction are the metrics from each class we compare in our study, while the predicted feature — is the post-release defects. The linear regression being the simplest model is reported to closely approximate the defect prediction function D'Ambros et al. (2012). The generalized linear regression models are trained throughout our experiments, whether comparing the approaches for classification, ranking, and ranking with effort consideration. They are even trained within the feature selection procedure to evaluate the performance of different feature subsets. Generalized Linear Models (GLMs) are used for predicting categorical outcome in classification as well as count outcome in regression Zhao (2012).

5.4 Five-fold cross-validation

We perform stratified five-fold repeated cross-validation experiments, i.e., we split the dataset into five folds, using four folds (80% of the dataset) as a training set to train the prediction model and the remaining fold (20% of the dataset) as a testing set to evaluate the model performance. Each of the five folds is used once as a testing set. The five-fold experiment is repeated 10 times to get a robust evaluation of the prediction model. To maintain consistency in the distribution of the predicted feature in each fold with the distribution of the entire dataset stratified cross-validation is used.

6 Experiment 2: cross-project defect prediction context

In the CPDP experiment, for each dataset reserved as a target project dataset, the remaining datasets combined constitute what we call multi-source training data set (TDS). For comparing the 25 defect prediction approaches listed in Predictor column of Table 1 in the cross-project prediction context a methodology similar to Experiment 1 (refer to Section 5) is followed. However, for each target project dataset, the feature selection and final

model training are done over the multi-source training data set. Moreover, the same data folds from each target project dataset that are used for model validation in the final step of Experiment 1 are used as testing sets in the performance evaluation of cross-project prediction models.

7 Results and discussion of defect prediction approaches

In the first evaluation we rank the approaches — across several project data sets — adhering a stringent statistical methodology. Towards this goal, we follow the approach of D'Ambros et al. (2012), Jiang et al. (2008) and Lessmann et al. (2008).

For every prediction approach we determine the ranking on each project in terms of performance metrics (AUC, $p_{popt}, p_{peffort}$). Moreover, we find the average rank (AR) of each prediction approach on all project data sets. Next, to ascertain if the performance differences with regard to AR are statistically significant (H_0 : all prediction approaches perform equally) we make use of the Friedman test with the Nemenyi's post hoc test Calvo and Santaf'e (2015), Demšar (2006) on the rankings.

In Tables 3, 4 and 5 (within-project performance) Tables 6, 7, and 8 (cross-project performance), we report the mean of performance metrics (AUC, p_{opt} and p_{effort}) from cross-validation experiments on each project dataset and AR across all project datasets. These tables are structured similarly. Each prediction approach is shown on a separate row, where the mean performance across five folds from ten runs on the subject datasets is represented

Table 3 Within-project AUC values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.822**	**0.800**	**0.776**	**0.733**	**0.818**	2.4
NFIX-ONLY	0.672	0.487	0.648	0.466	0.617	24.4
NR	0.764	0.555	**0.741**	0.674	0.725	18
NFIX+NR	0.767	0.576	**0.742**	0.686	0.725	16.6
BUG-CAT	0.784	0.718	0.707	0.694	0.771	16.7
BUG-FIX	0.771	0.572	0.695	0.647	0.771	18.3
CK+OO	**0.803**	**0.775**	**0.744**	**0.738**	0.743	7
CK	0.784	0.751	0.726	**0.729**	0.687	13
OO	**0.802**	**0.772**	**0.743**	**0.730**	0.726	8.2
LOC	0.679	0.464	0.652	0.483	0.507	24.4
HCM	**0.802**	0.489	**0.740**	0.695	0.755	15.6
WHCM	**0.799**	0.567	0.727	**0.722**	0.750	14.8
EDHCM	0.745	0.564	0.731	0.696	**0.777**	16.8
LDHCM	0.750	0.557	**0.739**	**0.714**	0.772	14.4
LGDHCM	0.757	0.469	**0.748**	**0.705**	0.777	14.4
CHU	**0.823**	0.729	0.687	0.699	0.760	14.2
WCHU	**0.834**	0.736	0.732	**0.709**	0.767	10.4
LDCHU	**0.821**	0.725	0.733	**0.727**	**0.796**	7.8
EDCHU	**0.797**	0.738	0.732	**0.716**	**0.790**	10.8
LGDCHU	**0.823**	0.708	0.735	**0.721**	**0.792**	9
HH	**0.833**	0.740	0.735	**0.712**	0.764	9.8
HWH	**0.835**	0.721	0.713	**0.712**	0.766	12
LDHH	**0.820**	0.724	0.735	**0.724**	**0.793**	8.4
EDHH	**0.799**	0.744	0.723	**0.722**	**0.804**	9.4
LGDHH	**0.828**	0.720	**0.740**	**0.714**	**0.793**	8.2

Table 4 Within-project p_{opt} values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.892**	**0.843**	**0.901**	**0.811**	**0.875**	3
NFIX-ONLY	0.783	0.604	0.837	0.567	0.716	24.4
NR	**0.853**	0.656	**0.884**	0.767	0.797	19.4
NFIX+NR	**0.855**	0.667	**0.886**	0.774	0.794	18.4
BUG-CAT	**0.874**	0.763	**0.875**	**0.794**	**0.850**	15.9
BUG-FIX	**0.870**	0.670	**0.879**	0.771	**0.850**	16.5
CK+OO	**0.882**	**0.818**	**0.897**	**0.821**	0.784	7.8
CK	**0.875**	0.791	**0.888**	0.762	0.754	16.6
OO	**0.879**	**0.815**	**0.897**	**0.818**	0.769	9.2
LOC	0.839	0.708	**0.867**	0.639	0.533	22
HCM	**0.876**	0.582	**0.885**	0.781	0.813	18.2
WHCM	**0.883**	0.642	**0.878**	**0.813**	0.819	14.6
EDHCM	0.839	0.647	**0.878**	**0.799**	**0.837**	17.8
LDHCM	0.839	0.641	**0.890**	**0.810**	**0.832**	15.8
LGDHCM	0.834	0.529	**0.890**	**0.797**	**0.832**	18
CHU	**0.879**	**0.802**	**0.878**	**0.788**	**0.834**	13.2
WCHU	**0.883**	0.798	**0.892**	**0.802**	**0.836**	10
LDCHU	**0.887**	0.788	**0.892**	**0.823**	**0.861**	6
EDCHU	**0.873**	0.799	**0.886**	**0.819**	**0.867**	9
LGDCHU	**0.880**	0.773	**0.897**	**0.805**	**0.856**	9.8
HH	**0.886**	**0.803**	**0.896**	**0.802**	0.829	9
HWH	**0.885**	0.777	**0.878**	**0.809**	**0.833**	12.4
LDHH	**0.893**	0.796	**0.901**	**0.821**	**0.863**	4.2
EDHH	**0.877**	0.801	**0.887**	**0.817**	**0.871**	8
LGDHH	**0.894**	0.777	**0.903**	**0.810**	**0.862**	5.8

Table 5 Within-project p_{effort} values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	0.797	**0.751**	**0.928**	**0.700**	**0.854**	10.8
NFIX-ONLY	0.721	**0.728**	0.876	0.642	0.808	18.4
NR	0.750	0.611	0.847	0.675	0.819	20.2
NFIX+NR	0.759	0.701	0.870	0.681	0.815	18
BUG-CAT	0.792	0.692	0.912	**0.731**	**0.863**	12.1
BUG-FIX	0.772	0.631	0.832	0.670	**0.863**	18.9
CK+OO	**0.805**	0.704	**0.961**	**0.737**	0.820	8.8
CK	**0.800**	0.653	**0.950**	**0.736**	0.824	11.8
OO	0.777	0.704	**0.958**	**0.723**	0.805	12.6
LOC	0.668	0.444	0.745	0.448	0.352	25
HCM	0.783	0.585	0.835	0.685	0.806	18.6
WHCM	0.766	0.562	0.806	0.677	0.803	22.4
EDHCM	**0.804**	0.643	0.848	**0.718**	0.770	15.6
LDHCM	**0.802**	0.641	0.837	**0.714**	0.780	16.4
LGDHCM	0.793	0.578	0.834	**0.701**	0.803	18.4
CHU	**0.816**	**0.745**	**0.953**	**0.704**	**0.869**	6.4
WCHU	**0.819**	**0.731**	**0.953**	0.681	**0.866**	9.8
LDCHU	**0.823**	**0.731**	**0.947**	**0.709**	**0.868**	7.2
EDCHU	**0.819**	**0.749**	**0.936**	0.682	**0.867**	9
LGDCHU	**0.829**	**0.727**	**0.951**	0.682	**0.865**	9.2
HH	**0.820**	**0.744**	**0.971**	0.684	**0.861**	7.4
HWH	**0.840**	**0.722**	**0.954**	0.682	**0.868**	7.4
LDHH	**0.818**	**0.736**	**0.966**	0.697	**0.863**	8.2
EDHH	**0.824**	**0.738**	**0.950**	**0.737**	**0.866**	5.4
LGDHH	**0.825**	**0.747**	**0.969**	0.684	**0.859**	7

Table 6 Cross-project AUC values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.805**	0.664	**0.743**	0.644	**0.778**	8
NFIX-ONLY	0.672	0.493	0.648	0.478	0.617	24
NR	0.764	0.555	**0.741**	0.674	0.725	14.2
NFIX+NR	**0.767**	0.546	**0.742**	0.680	0.725	14
BUG-CAT	0.754	0.604	0.692	0.629	**0.771**	14.5
BUG-FIX	**0.771**	0.572	0.695	0.647	**0.771**	12.5
CK+OO	0.758	0.698	0.591	**0.732**	0.706	12.8
CK	0.744	**0.736**	0.690	**0.715**	0.666	12.2
OO	0.741	**0.709**	0.700	**0.722**	0.678	12
LOC	0.679	0.516	0.652	0.510	0.449	23.4
HCM	**0.802**	0.489	**0.740**	0.695	**0.755**	12.2
WHCM	**0.799**	0.567	**0.727**	**0.722**	0.750	9.4
EDHCM	0.745	0.564	**0.731**	**0.696**	**0.777**	12
LDHCM	0.750	0.557	**0.739**	**0.714**	**0.772**	10.4
LGDHCM	0.757	0.531	**0.748**	**0.705**	**0.777**	9.4
CHU	0.748	0.634	0.579	0.689	**0.761**	16.8
WCHU	0.751	0.698	0.675	0.691	**0.765**	13.4
LDCHU	**0.767**	**0.706**	0.677	**0.706**	**0.793**	7
EDCHU	0.711	**0.715**	0.672	**0.702**	**0.786**	11
LGDCHU	0.671	0.682	0.693	**0.701**	**0.774**	12.8
HH	0.745	0.699	0.673	0.687	**0.755**	15
HWH	**0.790**	0.631	0.668	**0.696**	0.747	13.6
LDHH	0.749	**0.713**	0.682	**0.699**	**0.777**	10
EDHH	0.714	**0.720**	0.658	**0.705**	**0.769**	12.6
LGDHH	0.758	0.671	**0.710**	0.683	**0.767**	11.8

Table 7 Cross-project p_{opt} values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.884**	0.747	**0.898**	**0.799**	**0.832**	5.2
NFIX-ONLY	0.783	0.604	0.837	0.594	0.716	23.8
NR	**0.853**	0.656	**0.884**	0.767	0.797	15.4
NFIX+NR	**0.854**	0.653	**0.886**	**0.774**	0.793	14.4
BUG-CAT	**0.854**	0.689	**0.869**	0.720	**0.850**	13.1
BUG-FIX	**0.870**	0.670	**0.879**	0.771	**0.850**	10.1
CK+OO	**0.859**	**0.789**	0.796	**0.794**	0.763	12.8
CK	**0.842**	**0.775**	**0.881**	0.725	0.709	14.4
OO	**0.852**	0.736	**0.879**	0.769	0.743	15.2
LOC	0.839	0.708	**0.867**	0.657	0.592	18.6
HCM	**0.876**	0.582	**0.885**	**0.781**	0.813	12.8
WHCM	**0.883**	0.642	**0.878**	**0.813**	0.819	9.8
EDHCM	0.839	0.647	**0.878**	**0.799**	**0.837**	12.2
LDHCM	0.839	0.641	**0.890**	**0.810**	**0.832**	11
LGDHCM	0.834	0.618	**0.890**	**0.797**	**0.832**	13
CHU	0.832	0.658	0.771	**0.783**	0.799	19.2
WCHU	**0.871**	0.744	0.849	**0.785**	**0.836**	10.4
LDCHU	**0.862**	**0.771**	0.844	**0.808**	**0.869**	7.2
EDCHU	**0.857**	**0.766**	0.830	**0.798**	**0.865**	9.4
LGDCHU	0.795	0.720	**0.857**	0.752	**0.835**	16
HH	**0.857**	**0.758**	0.851	**0.781**	**0.828**	12.6
HWH	**0.842**	0.670	0.848	**0.776**	0.814	16.8
LDHH	**0.874**	**0.770**	**0.862**	**0.793**	**0.829**	9.4
EDHH	**0.856**	**0.772**	**0.853**	**0.798**	**0.831**	10
LGDHH	0.836	**0.752**	**0.865**	**0.801**	**0.826**	12.2

Table 8 Cross-project p_{effort} values of all predictors over all systems

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.797**	0.469	0.851	0.685	**0.854**	10.4
NFIX-ONLY	0.721	**0.728**	0.876	0.664	0.808	13
NR	0.750	0.611	0.847	0.675	0.819	14.4
NFIX+NR	0.722	0.615	0.848	0.650	0.815	18.4
BUG-CAT	**0.792**	0.668	**0.914**	**0.731**	0.796	8.4
BUG-FIX	**0.772**	0.631	0.832	0.670	**0.863**	11.6
CK+OO	0.745	0.619	0.857	0.650	0.726	17.2
CK	**0.800**	0.626	**0.950**	0.651	**0.824**	9.6
OO	0.746	0.621	0.859	0.579	0.730	18
LOC	0.668	0.444	0.745	0.439	0.310	25
HCM	**0.783**	0.585	0.835	0.685	0.806	14.4
WHCM	**0.766**	0.562	0.806	0.677	0.803	16.4
EDHCM	**0.804**	0.643	0.848	**0.718**	0.770	11
LDHCM	**0.802**	0.641	0.837	**0.714**	0.780	11.8
LGDHCM	**0.793**	0.626	0.834	**0.701**	0.803	13.2
CHU	0.732	0.691	0.859	0.668	**0.834**	12.6
WCHU	0.731	0.689	0.868	0.607	**0.832**	14.2
LDCHU	0.733	0.679	0.859	0.604	**0.842**	13.2
EDCHU	0.732	0.655	0.886	0.611	**0.842**	12.4
LGDCHU	0.732	0.628	0.891	0.605	**0.836**	14.6
HH	0.735	**0.697**	**0.920**	0.675	**0.836**	8.6
HWH	0.737	0.656	**0.908**	0.609	**0.841**	11.8
LDHH	0.738	0.686	**0.919**	0.675	**0.843**	8
EDHH	0.737	**0.704**	**0.924**	0.594	**0.843**	8.8
LGDHH	0.735	**0.700**	**0.920**	0.675	**0.841**	8

in the first five cells. The best-performing approaches are highlighted by showing the mean performance values above 95% of the best value in bold font. The AR of the approaches over the subject datasets is shown in the last cell of each row. The approaches with overall good performance (AR values ≤ 10, or top 40% of the ranking) are shown with a shaded background.

7.1 Results of within-project defect prediction

AUC in within-project classification The top-ranked approaches are MOSER (2.4), CK-OO (7), LDCHU (7.8), OO and LGDHH (tied 8.2), and LDHH (8.4). Thenceforth, AR drops slowly until approaches ranked around 18, where it hops down to the worst rank 24.4 in LOC and NFIX-ONLY. In general, the process metrics, regular source code metrics, churn of source code, and entropy of source code metrics perform remarkably well. Defect metrics perform next. The entropy of change metrics (AR 14.4 to 16.8) performs very poorly. A probable reason for the bad performance of the entropy of change metrics is that each approach from the set corresponds to a single metric, which on its own may not have sufficient explanatory power to distinguish the files as clean or defective. Likewise, approximations of process, or source code metrics that involve single or a few metrics perform quite badly.

p_{opt} in within-project ranking The top performers are MOSER (3), LDHH (4.2), LGDHH (5.8), LDCHU (6) and CK-OO (7.8) after that most of the entropy and churn of source code metrics. The defect metrics have comparatively bad ranks (15.9 and 16.5). As pointed

out before sets of approaches that use few metrics — entropy of change metrics, approximation of process, or source code metrics performed quite badly (14.6 or below). the worst performers are NFIX-ONLY (24.4) and LOC (22).

p_{effort} in within-project ranking The top performers are EDHH (5.4), CHU (6.4), LGDHH (7), LDCHU (7.2), HH and HWH (tied 7.4), LDHH (8.2), and CK-OO (8.8). Almost all of the churn and entropy of source code metrics give a good predictive performance. All the ranks below 10 are taken by one of these, except for the rank 8.8 that is occupied by CK-OO (placing it 8th in the rank). The worst performers are LOC (25), WHCM (22.4), NR (20.2), BUG-FIX (18.9), HCM (18.6) NFIX-ONLY (18.4), NFIX+NR (18). The general trend of approaches utilizing fewer metrics performing worse is observed. The process metrics perform worst than the effort-unaware scenario ranked 10.8 for p_{effort} and 3 for p_{opt}. These results contradict those of Mende and Koschke (2009), Kamei (2010), and D'Ambros et al. (2012).

7.2 Discussion of within-project defect prediction rankings

The null hypothesis of all approaches performing equally is rejected by the Friedman test when AUC, p_{opt}, and p_{effort} are considered. However, Nemenyi's post hoc test could only separate the very best performer MOSER from the worst NFIX-ONLY, and LOC for the AUC. In Fig. 1, we present the findings of Nemenyi's test using Demsar's significance diagrams Demšar (2006). All classifiers that are not connected with a horizontal line can be seen as performing significantly differently. Similarly, for p_{opt} the Nemenyi's test could separate the best performers MOSER and LDHH from the worst LOC, and NFIX-ONLY. Considering p_{effort} the EDHH, CHU, LGDHH, LDCHU, HH, and HWH, are statistically better than LOC. The statistical tests are performed at a significance level of 0.05.

LOC is the worst approach under all evaluation criteria. And the NFIX-ONLY is the worst approach for classification and ranking without effort consideration.

A different ordering of the approaches is generated by each task, which depicts that each problem has different characteristics.

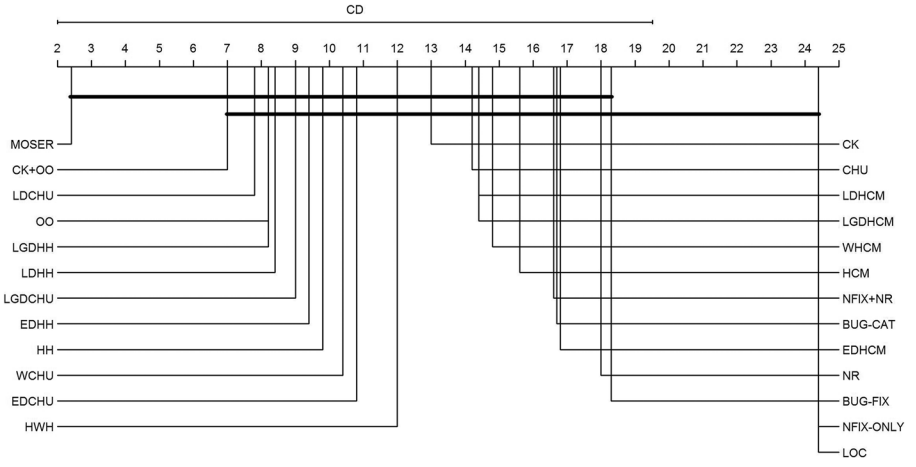


Fig. 1 Demsar's significance diagrams for AUC in within-project context

In general, the best performers for classification are process metrics and product metrics after that the churn of source code and entropy of source code metrics. The process metrics also perform best for ranking without effort consideration, followed by the entropy of source code, the churn of source code metrics, and product metrics. In ranking with effort consideration, the entropy and churn of source code perform best followed by product and process metrics. The defect metrics and other single metric approaches (entropy of changes, number of revisions, lines of code) performed worst in all three scenarios.

7.3 Results of cross-project defect prediction

AUC in cross-project classification The top-ranked approaches are LDCHU (7), MOSER (8), WHCM and LGDHCM (tied at 9.4), and LDHH (10). Then, AR goes down slowly until the approaches ranked around 16, where it abruptly drops to the worst rank 24 in NFIX-ONLY and 23.4 in LOC. In general, the churn of source code metrics, process metrics, the entropy of changes, and the entropy of code metrics perform quite well. Regular source code metrics perform next. Defect metrics perform very poorly. Apart from the entropy of changes, the general trend of approaches featuring few metrics performing poorly is observed. Compared to within-project predictions among the top-ranked approaches only the LDCHU, MOSER, and LDHH feature in top performers. And the worst-ranked approaches are the same for within-project and cross-project contexts. That means the process metrics, the churn of code metrics and the entropy of code metrics are good approaches to consider for defect prediction in the classification context. And NFIX-ONLY and LOC are generally very poor approaches.

p_{opt} in cross-project ranking The top performers are MOSER (5.2), LDCHU (7.2), EDCHU and LDCHU (tied at 9.4), WHCM (9.8), and EDHH (10). The worst performers are NFIX-ONLY (23.8), CHU (19.2) and LOC (18.6). The general trend of approaches with few metrics performing poorly is continued except for WHCM. Compared to within-project predictions among the top performers only the MOSER, LDCHU, EDCHU, LDHH and EDHH feature in top performers. And the worst performers (NFIX-ONLY and LOC) from within-project predictions are among the worst three approaches in cross-project predictions. Therefore, we conclude process metrics, the churn of code metrics, and entropy of code metrics are generally good approaches to consider for defect prediction in the ranking context. And NFIX-ONLY and LOC are the worst approaches.

p_{effort} in cross-project ranking The top ranks are achieved by LDHH and LGDHH (tied at rank 8), BUG-CAT (8.4) HH (8.6), EDHH (8.8), and CK (9.6). Except for HWH, all the entropy of source code metrics achieve an average rank below 10. Then, the performance goes down slowly until the approaches ranked around 18.4, where the AR suddenly drops to the worst AR 25 in LOC. Except for BUG-CAT the general trend that approaches with only a few metrics perform poorly is continued. The entropy of source code metrics LDHH achieves an AR below 10 in both the within-project and cross-project context under all the three performance indicators of AUC, p_{opt}, and p_{effort}. The LOC metric always features among the worst two approaches. Therefore based on the experimental results we suggest, always include the LDHH metrics and exclude the LOC metrics in model training for the defect prediction task.

7.4 Discussion of cross-project prediction rankings

In the case of cross-project defect prediction, at a significance level of 0.05, no distinction of average ranks could be made. However, at a significance level of 0.1, the null hypothesis of all approaches performing equally is rejected by the Friedman test when AUC, p_{opt}, and p_{effort} are considered. The Nemenyi's post hoc test only separates the very best performer LDCHU from the worst LOC, and NFIX-ONLY for the AUC. In Fig. 2, we present the findings of Nemenyi's test using Demsar's significance diagrams (2006). Similarly, for p_{opt} the Nemenyi's test separates the best performers MOSER and LDCHU against the worst NFIX-ONLY metric. Considering p_{effort} the LDHH, LGDHH, BUG-CAT, and HH are statistically better than LOC.

LOC is the worst approach for classification and ranking with effort consideration. And NFIX-ONLY is the worst for classification and ranking without effort consideration.

Different ordering of the approaches is generated by each task and compared to within-project predictions ordering of the approaches is also different that means that each problem has different characteristics and conclusions from within-project studies should not be generalized to cross-project predictions.

In general, the best performers for classification are the churn of source code followed by process metrics, the entropy of changes, and the entropy of source code metrics. The defect metrics and other single metric approaches (except for entropy of changes) performed worst for classification. For ranking without effort consideration the process metrics perform best, followed by the churn of source code, entropy of source code, entropy of changes, and product metrics. The defect metrics and other single metric approaches (except for entropy of changes) performed worst for ranking without effort consideration as well. In ranking with effort consideration the entropy of source code and defect metrics perform best followed by product and process metrics. The churn of source code, the entropy of change metrics and other single metric approaches (number of revisions, lines of code) perform worst.

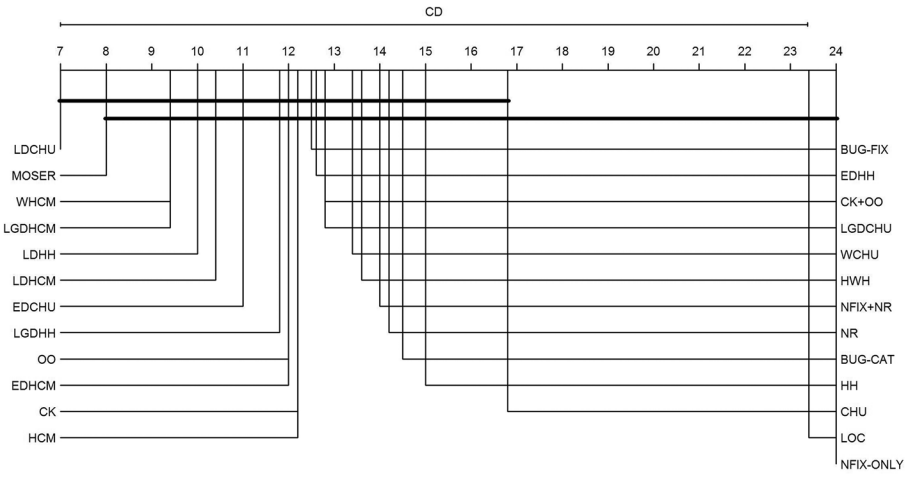


Fig. 2 Demsar's significance diagrams for AUC in cross-project context

8 Statistical significance for defect prediction approaches

Motivated by D'Ambros et al. (2012), Menzies et al. (2010), our second set of evaluations aim to check the variability of the approaches across several runs and at determining statistically significant ranking of the approaches. To get an estimate of the likely variability of performance, for each defect prediction approach, we save the 50 results of cross-validation (last line in algorithm 1) for each performance measure. For each performance measure, we combine the 50 data points for each dataset generating a set of 250 data points that represent all five datasets. Then, the median, first quartile, and third quartile of each set of 250 data points for the defect prediction approaches are computed. Next, the approaches are sorted by their medians, and displayed visually through "mini boxplots". The first-third quartile range is represented by a bar and the median by a circle. Figure 3 shows the mini boxplot for all the approaches over all the five datasets for classification.

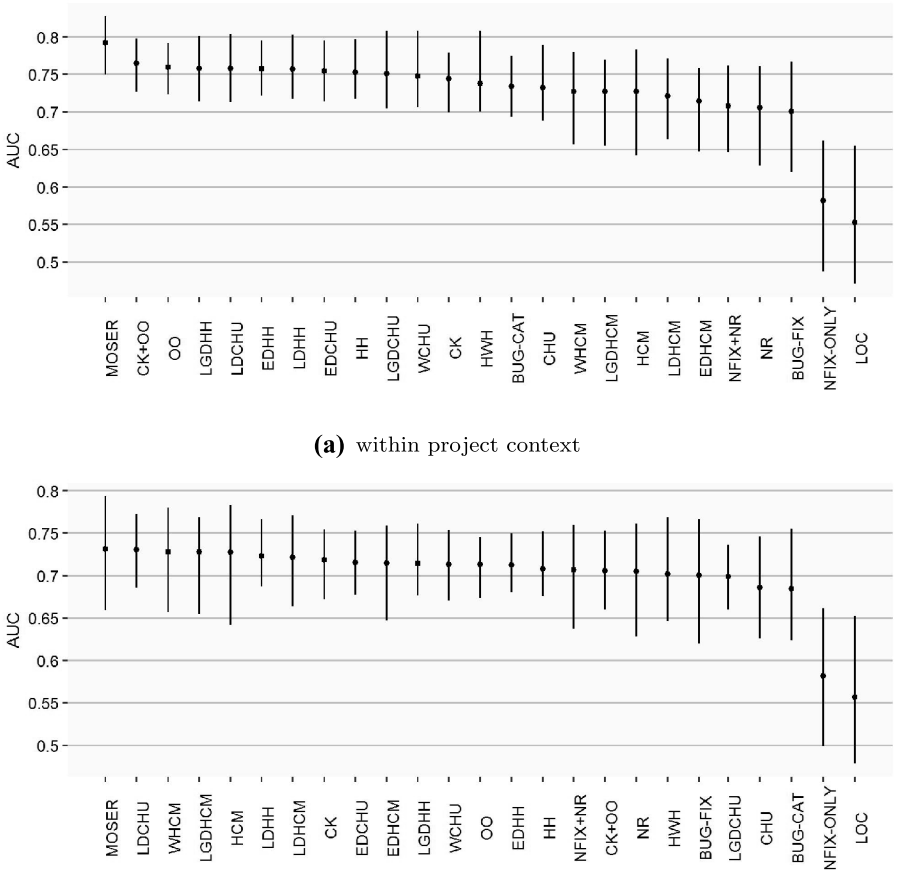
In within-project context for 75% of total 250 predictions over five datasets, the AUC, p_{opt} and p_{effort} values of the resultant top ranking approaches from the evaluations in Section 7.1 are greater than 0.7. That means the top ranking approaches are able to strongly predict the defects in within-project context.

In cross-project context for 75% of total 250 prediction over five datasets, the AUC, p_{opt} and p_{effort} values of the resultant top ranking approaches from the evaluations in Section 7.3 are greater than 0.65. The AUC and p_{effort} values are greater than 0.7 for 50% of total predictions and the p_{opt} value is greater than 0.7 for 75% of predictions. That means the top ranking approaches are at least able to feasibly predict the defects in cross-project context.

As shown in Fig. 3, in our experiments, many approaches perform similarly. Therefore, it is difficult to get a meaningful ranking. So, we apply the technique used by D'Ambros et al. (2012), Menzies et al. (2010) to compare the categories of approaches instead of individual approaches. To simplify the comparison we select the best representative from each category (one with the highest median). Menzies et al. (2010) and D'Ambros et al. (2012) generate a ranking by doing the Mann-Whitney U test on each successive duo of approaches. If the U test fails to reject the null hypothesis — that performance distributions are equal at 95% confidence level, then the two approaches have the same rank. To get a different rank with other approaches, the test when applied to all other approaches of equivalent rank must reject the null hypothesis. Starting with the top two approaches this procedure is repeated sequentially.

8.1 Finding statistical significance for within-project defect prediction

AUC in within-project classification For all systems, the classification results, including the AUC medians of selected approaches, and for every combination of approaches, if Mann-Whitney U test reject the null hypothesis that the two perform equally or not is shown in Table 9. The medians range from 0.728 to 0.793. The U test results show that the approach built on process metrics (MOSER–rank 1) outperforms source code metrics (CK+OO–rank 2), the entropy of code metrics (LGDHH–rank 2), the churn of code metrics (LDCHU–rank 2), and previous defects (BUG-CAT–rank 5), the entropy of changes (WHCM–rank 5) in classification with 95% confidence. One interesting fact is that the process metrics (MOSER) perform significantly better than all other approaches.



(a) within project context

(b) cross project context

Fig. 3 Mini boxplots for classification over all systems in within- and cross-project context

p_{opt} in within-project ranking For all systems, the results of ranking in terms of p_{opt} are shown in Table 10. The medians of all chosen approaches range from 0.836 to 0.866. Apart from the entropy of source code metrics where LGDHH is substituted by LDHH, the top performers in each category are identical with classification. The order generated

Table 9 Medians, results of Mann-Whitney U test and rank of approaches for all systems in within-project defect prediction context when approaches are compared for classification. From each group only the top performing approaches are compared in terms of *AUC*

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	CK+OO	LGDHH	LDCHU	BUG-CAT	WHCM	Rank
MOSER	0.793	yes	yes	yes	yes	yes	1
CK+OO	0.765		no	no	yes	yes	2
LGDHH	0.758			no	yes	yes	2
LDCHU	0.758				yes	yes	2
BUG-CAT	0.734					no	5
WHCM	0.728						5

Table 10 Medians, results of Mann-Whitney U test and rank of approaches for all systems within-project defect prediction context when approaches are compared for ranking. From each group only the top performing approaches are compared in terms of p_{opt}

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	LDCHU	LDHH	CK+OO	WHCM	BUG-CAT	Rank
MOSER	0.866	yes	no	yes	yes	yes	1
LDCHU	0.861		no	no	yes	yes	1
LDHH	0.860			yes	yes	yes	1
CK+OO	0.854				yes	yes	4
WHCM	0.842					no	5
BUG-CAT	0.836						5

by sorting the approaches by decreasing medians is different. However, the process metrics (MOSER) still occupy the first position. The Mann-Whitney U test results are different from classification. The process metrics (MOSER–rank 1), the churn of code metrics (LDCHU–rank 1), and entropy of code metrics (LDHH–rank 1) are better than source code metrics (CK+OO–rank 4), and the entropy of changes (WHCM–rank 5), previous defects (BUG-CAT–rank 5). The worst performers (entropy of changes and previous defects) are the same as classification.

p_{effort} in within-project ranking For all systems, the results of effort-aware ranking with regard to p_{effort} are shown in Table 11. The medians of all chosen approaches range from 0.758 to 0.819. The best performers per category are different from classification and ranking with p_{opt}. Overall worst is the approach based on the entropy of changes (HCM–rank 6), clearly underperforming other approaches in effort-aware ranking. In general, saying one approach is better than the other is difficult, as is clear by the same rank (rank 1) for five categories of approaches out of six. Therefore, we conclude getting a confident ranking with p_{effort} is more difficult than classification and ranking without effort consideration.

8.2 Discussion of statistical significance for within-project defect prediction

In general, for classification process metrics (rank 1) followed by the churn of source code metrics (rank 2), the entropy of source code metrics (rank 2), and source code metrics (rank 2) are the best performers from a statistical point of view. In ranking with p_{opt} process metrics (rank 1) tied in rank with the churn of source code metrics and entropy of source code metrics are the best performers. Effort-aware ranking with p_{effort} is a harder problem where five out of six categories of approaches tied in the first rank are un-distinguishable from a statistical significance point of view. The worst performers for effort-aware ranking are entropy of changes (rank 6) which is tied in worst rank (rank 5) with previous defects in ranking and classification.

Table 11 Medians, results of Mann-Whitney U test and rank of approaches for all systems in within-project defect prediction context when approaches are compared for effort-aware ranking. From each group only the top performing approaches are compared in terms of p_{effort}

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	LGDCHU	BUG-CAT	CK+OO	MOSER	HCM	Rank
HWH	0.819	no	no	no	no	yes	1
LGDCHU	0.808		no	no	no	yes	1
BUG-CAT	0.790			no	no	yes	1
CK+OO	0.788				no	yes	1
MOSER	0.785					yes	1
HCM	0.758						6

Table 12 Medians, results of Mann-Whitney U test and rank of approaches for all systems cross-project defect prediction context when approaches are compared for classification. From each group only the top performing approaches are compared in terms of *AUC*

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	LDCHU	WHCM	LDHH	CK	BUG-FIX	Rank
MOSER	0.731	no	no	no	yes	yes	1
LDCHU	0.731		no	no	yes	yes	1
WHCM	0.728			no	no	yes	1
LDHH	0.723				yes	yes	1
CK	0.719					yes	1
BUG-FIX	0.700						6

The overall best performers are the process metrics, the churn of source code, and entropy of source code metrics. And the overall worst performers are entropy of change metrics.

8.3 Finding statistical significance for cross-project defect prediction

AUC in cross-project classification The results of the Mann-Whitney U test for classification (AUC) in cross-project context are shown in Table 12. The medians of all chosen approaches range from 0.700 to 0.731. The sorting of the approaches by decreasing medians generates an order in which the process metrics (MOSER) are placed at the first position and the BUG-FIX is placed at the last position. MOSER, LDCHU, WHCM, LDHH, and CK are un-distinguishable from a statistical significance point of view because all the five approaches get a similar rank (rank 1). The defect metrics (BUG-FIX) get the worst rank (rank 6). In general, it is difficult to get a confident ranking with AUC in the cross-project context.

p_{opt} in cross-project ranking The results of the Mann-Whitney U test for ranking without effort consideration (p_{opt}) in cross-project context are shown in Table 13. The medians of all chosen approaches range from 0.806 to 0.846. Apart from source code metrics where CK is substituted by CK+OO, the top performers in each category are similar as classification. The sorting of the approaches by decreasing medians generates a dissimilar order. However, the process metrics (MOSER) are again placed at the first position. MOSER, WHCM, LDCHU, BUG-FIX and LDHH are un-distinguishable from a statistical significance point of view because all the five approaches get a similar rank (rank 1). The product metrics (CK+OO) get the worst rank (rank 6). In general, it is difficult to get a confident ranking with p_{opt} in cross-project context.

Table 13 Medians, results of Mann-Whitney U test and rank of approaches for all systems in cross-project defect prediction context when approaches are compared for ranking. From each group only the top performing approaches are compared in terms of p_{opt}

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	WHCM	LDCHU	BUG-FIX	LDHH	CK+OO	Rank
MOSER	0.846	no	no	yes	no	yes	1
WHCM	0.842		no	no	no	yes	1
LDCHU	0.838			no	no	yes	1
BUG-FIX	0.834				no	yes	1
LDHH	0.826					yes	1
CK+OO	0.806						6

p_{effort} in cross-project ranking The results of the Mann-Whitney U test for effort-aware ranking (p_{effort}) in cross-project context are shown in Table 14. The medians of all chosen approaches range from 0.744 to 0.780. The best performers are different from classification and ranking with p_{opt}. All the six approaches get a rank of 1. That means from a statistical significance point of view no distinction can be made between the approaches. Therefore, we conclude it is difficult to get a confident ranking with p_{effort} in the cross-project context.

8.4 Discussion of statistical significance for cross-project defect prediction

For classification in the cross-project context, the difference between process metrics, churn of source code metrics, entropy of change metrics, entropy of source code metrics, and regular source code metrics is not significant. These approaches perform significantly better than previous defect metrics. Similarly, for ranking without effort consideration only regular source code metrics perform significantly different than other top five approaches (process metrics, the entropy of change metrics, churn of source code, previous defects, the entropy of source code metrics). For ranking with effort consideration, no significant difference is found between the approaches and the experiments are unable to produce a ranking of significance.

The overall best performers are the process metrics, the churn of source code, and entropy of code metrics, and entropy of change metrics. For classification, previous defects perform worse, and for ranking without effort considering source code metrics perform worse. For ranking with effort consideration, no significant distinction could be made between the performance.

9 Experiment 3: Effect of class imbalance on the ranking of different approaches

9.1 Methodology

To check the effect of the data imbalance on the classification performance we use a data balancing technique, Border Line Synthetic Minority Oversampling (BLSMOTE) for creating synthetic minority defective class instances in the training dataset. Presuming borderline instances are most likely to be misclassified, Han et al. (2005) had proposed BLSMOTE to create synthetic minority class instances from the borderline minority class instances. BLSMOTE is reported to have stable, and good performance Bennin et al.

Table 14 Medians, results of Mann-Whitney U test and rank of approaches for all systems in cross-project defect prediction context when approaches are compared for effort-aware ranking. From each group only the top performing approaches are compared in terms of p_{effort}

		Mann-Whitney U-test reject null hypothesis					
Predictor	median	LDHCM	MOSER	CK	EDHH	CHU	Rank
BUG-CAT	0.780	yes	yes	no	yes	yes	1
LDHCM	0.771		no	no	no	no	1
MOSER	0.765			no	no	no	1
CK	0.764				no	no	1
EDHH	0.757					no	1
CHU	0.744						1

(2019) compared to other data balancing techniques in software defect prediction. It is due to the reasons cited above we use BLSMOTE for data balancing.

For evaluating the effect of class imbalance on the ranking of 25 defect prediction approaches listed in Predictor column of Table 1 a methodology similar to Experiment 1 (refer to Section 5) for WPDP and similar to Experiment 2 (refer to Section 6) for CPDP is followed. However, for each target project dataset, final model training is done over the balanced training data set generated through BLSMOTE. Moreover, the same data folds from each target project dataset that are used for model validation in the final step of Experiment 1 and Experiment 2 are used as testing sets in the performance evaluation of the models trained on BLSMOTE balanced datasets. The average ranking (AR) based on AUC performance over the five datasets is used to evaluate the ranking of the selected approaches.

9.2 Results and discussion

From the experimental results reported in Table 15, for both within-project and cross-project contexts we do not see much of the difference in the Average Ranking of the approaches when the models are trained on the original imbalanced datasets and when the models are trained on the datasets balanced through BLSMOTE technique. In general, the best performers for classification in within-project context (i.e., process metrics, product metrics, the churn of source code and the entropy of source code metrics) and in cross-project context (i.e., process metrics, the entropy of changes, and

Table 15 Average Ranking of defect prediction approaches on imbalanced and balanced datasets

	WPDP Average ranking (AR)		*CPDP Average ranking (AR)*	
Predictor	**Before BLSMOTE**	**After BLSMOTE**	**Before BLSMOTE**	**After BLSMOTE**
MOSER	2.4	3.8	8	6.8
NFIX-ONLY	24.4	24.2	24	23.8
NR	18	19	14.2	14.4
NFIX+NR	16.6	17.4	14	13.6
BUG-CAT	16.7	17.5	14.5	13.1
BUG-FIX	18.3	17.9	12.5	13.1
CK+OO	7	6	12.8	12
CK	13	13.8	12.2	16
OO	8.2	11	12	11
LOC	24.4	23.8	23.4	23
HCM	15.6	16.8	12.2	11.4
WHCM	14.8	15.2	9.4	8.6
EDHCM	16.8	16.2	12	11.6
LDHCM	14.4	15.6	10.4	10.2
LGDHCM	14.4	15	9.4	10
CHU	14.2	13.6	16.8	18.2
WCHU	10.4	11.8	13.4	12.8
LDCHU	7.8	6.6	7	12.4
EDCHU	10.8	7	11	10.8
LGDCHU	9	10	12.8	11
HH	9.8	8.4	15	13.8
HWH	12	11.2	13.6	13.8
LDHH	8.4	7.2	10	8.6
EDHH	9.4	8.4	12.6	14.4
LGDHH	8.2	7.6	11.8	10.6

the entropy of source code metrics) are same when imbalanced and balanced datasets are used for training. Similarly, the worst performers for classification (i.e., LOC and NFIX-ONLY) are also same when imbalanced and balanced datasets are used for training. For the remaining approaches also no major difference exists in the Average Rankings. The Average Rankings of the approaches are more or less same when the resultant balanced datasets from the BLSMOTE are used for model training as compared to using the original imbalanced datasets. Therefore, we conclude the data imbalance does not have much effect on the ranking of the evaluated approaches.

10 Threats to validity

Construct validity Threats to construct validity are related to the bias of data utilized in model training and metrics employed to assess the prediction performance. Since we use a public data set, we cannot directly validate the quality of data. However, D'Ambros et al. D'Ambros et al. (2010) describe in detail the procedure they used to mine different data repositories to create the Bug Prediction Dataset, which makes us count on the validity of the dataset used in our experiments. We evaluate the approaches in three different evaluation scenarios (classification, ranking, ranking with effort consideration).

Statistical conclusion validity To check if the performance difference between each approach is significant we perform Nemenyi's post hoc test. The test only separates the extremely best from the extremely worst performers. Due to its conservative behavior, the test has a higher possibility that it will fail to reject the null hypothesis of approaches performing equally. Therefore, we test the performance difference between duo of approaches with the Mann-Whitney U test at a 95% confidence level.

Internal validity Threats to the internal validity relate to the data transformation used, feature selection methods used, and the training algorithms used for the experiments. The software metrics measured at different scales contribute differently in model training and may result in biasing the model. Therefore, feature scaling (Min-Max scaling) is applied to the datasets. The multicollinearity and a large number of features in model training results in model overfitting. To address the problem of multicollinearity and use only informative features in model training a feature selection technique (sequential floating forward selection) is used on datasets before model training.

External validity Threats to external validity are related to the risks of generalizing the findings of a study to other contexts and datasets. The controversial results in different scenarios about the performance of defect prediction approaches presented in this article show how results obtained in a certain context and evaluated with a particular evaluation metric are problematic to generalize to different contexts, and evaluation metrics. In our experiments, we use data about five open-source projects from Bug Prediction Dataset. The replication of the study on other datasets is required to generalize the findings of the study.

11 Conclusions

The driving factor for defect prediction is resource allocation. Given an accurate estimate of the distribution of defects across software components, the quality assurance effort can be prioritized towards the problematic parts of the system. Many approaches differing in the data sources used, training methods, and evaluation methods have been developed to predict the defects in software components. The earlier studies compare the data sources to only a few other data sources, evaluate them using distinct performance indicators without any effort consideration, or compare them only in the within-project context.

We evaluated a set of representative defect prediction approaches from the literature in the within-project and cross-project context in three different scenarios: classification, ranking, and ranking with effort consideration. In the within-project context, for classification, process metrics followed by product metrics, the churn of source code and entropy of source code metrics perform the best. A subsequent evaluation identified, process metrics significantly outperform the other three approaches. The process metrics also perform best for ranking without effort consideration, followed by the entropy of source code, the churn of source code metrics, and product metrics. The subsequent evaluation found the difference between the top three not significant. In ranking with effort consideration, the entropy and churn of source code perform best, followed by product and process metrics. The subsequent evaluation did not find any significant difference.

In cross-project context, for classification, churn of source code succeeded by process metrics, the entropy of changes, the entropy of source code metrics, and product metrics perform the best. A subsequent evaluation could not find any significant difference in the top performers. In ranking without effort consideration, the process metrics perform best, succeeded by the churn of source code, entropy of source code, and entropy of changes. A subsequent evaluation could not find any significant difference in the top performers. In ranking with effort consideration, the entropy of source code and defect metrics perform best. The subsequent evaluation found no significant difference between any of the approaches.

In general, under all three evaluation scenarios we observe that the process metrics, the churn of source code, and entropy of source code perform significantly better in within-project as well as cross-project contexts. The defect metrics and other single metric approaches (entropy of changes, number of revisions, lines of code, number of fixes) generally perform worse in all three scenarios. Moreover, in general, for both WPDP and CPDP contexts no vivid difference in the Average Ranking of the approaches is observed between the models trained on the original imbalanced datasets and models trained on the datasets balanced through BLSMOTE technique. Therefore, we conclude that the data imbalance does not have any substantial effect on the ranking of the evaluated approaches.

We suggest considering the process metrics, the churn of source code, and entropy of source code metrics in future defect prediction studies as predictors and taking a great deal of care when considering the single metric approaches (number of revisions, lines of code, number of fixes, etc.). We reckon the instance selection and transfer learning techniques used for alleviating the distribution mismatch between software projects over the last decade if used after careful selection of the predictor metrics in software defect prediction could improve the performance. In future, we suggest to study the effects of feature selection in combination with instance selection and transfer learning on the prediction performance.

We also observe that different ordering of approaches is generated by each evaluation scenario, in within-project as well as cross-project contexts. That means each problem has distinct characteristics. Therefore, conclusions from within-project defect prediction studies should not be generalized to cross-project defect predictions.

References

Agrawal, A., & Malhotra, R. (2019). Cross project defect prediction for open source software. *International Journal of Information Technology*.

Al Majzoub, H., Elgedawy, I., Akaydın, O., & KöseUlukök, M. (2020). Hcab-smote: A hybrid clustered affinitive borderline smote approach for imbalanced data binary classification.*Arabian Journal for Science and Engineering*, vol.45, no.4, pp.3205–3222.

Arisholm, E., Briand, L. C., & Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software, 83*, 2–17.

Arisholm, E., Briand, & L. C., Fuglerud, M. (2007). Data mining techniques for building fault-proneness models in telecom java software in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE.*

Barua, S., Islam, M. M., Yao, X., & Murase, K. (2014). MWMOTE-majority weighted minority oversampling technique for imbalanced data set learning. *IEEE Transactions on Knowledge and Data Engineering, 26,* 405–425.

Bashir, K., Li, T., Yohannese, C. W., & Yahaya, M. (2020). SMOTEFRIS-INFFC: Handling the challenge of borderline and noisy examples in imbalanced learning for software defect prediction. *Journal of Intelligent & Fuzzy Systems, 38,* 917–933.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22,* 751–761.

Bennin, K. E., Keung, J., Monden, A., Phannachitta, P., & Mensah, S. (2017). The significant effects of data sampling approaches on software defect prioritization and classification. *In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp.364–373, IEEE Press

Bennin, K. E., Keung, J. W., & Monden, A. (2019). On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering, 24*(2), 602–636.

Bennin, K. E., Keung, J., Phannachitta, P., Monden, A., & Mensah, S. (2017). Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. *IEEE Transactions on Software Engineering, 44*(6), 534–550.

Bennin, K. E., Tahir, A., MacDonell, S. G., & Börstler, J. (2022). An empirical study on the effectiveness of data resampling approaches for cross-project software defect prediction. *IET Software, 16*(2), 185–199.

Bhat, N. A., & Farooq, S. U. (2021a). An improved method for training data selection for cross-project defect prediction. *Arabian Journal for Science and Engineering*, pp. 1–16

Bhat, N. A., & Farooq, S. U. (2021b). *Local modelling approach for cross-project defect prediction*. Intelligent Decision Technologies: An International Journal.

Capretz, L. F., & Xu, J. (2008). An empirical validation of object-oriented design metrics for fault prediction. *Journal of computer science, 4*(7), 571.

Calvo, B., & Santaf'e, G. (2015). Scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems.R package version 0.2.3.

Çatal, Ç. (2016). The use of cross-company fault data for the software fault prediction problem. *Turkish Journal of Electrical Engineering & Computer Sciences, 24*(5), 3714–3723.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: synthetic minority oversampling technique. *Journal of artificial intelligence research, 16*, 321–357.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design.*IEEE Transactions on Software Engineering*, vol.20, pp.476–493

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. *In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp.31–41

D'Ambros, M., Lanza, M., & Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, vol.17, no.4, pp.531–577.

Dar, A. W., & Farooq, S. U. (2022). A survey of different approaches for the class imbalance problem in software defect prediction. *International Journal of Software Science and Computational Intelligence (IJSSCI), 14*(1), 1–26.

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

Fawcett, T. (2006). An introduction to roc analysis. *Pattern recognition letters, 27*(8), 861–874.

Felix, E. A., & Lee, S. P. (2017). Integrated Approach to Software Defect Prediction. *IEEE Access, 5*, 21524–21547.

Feng, S., Keung, J., Yu, X., Xiao, Y., & Zhang, M. (2021). Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction. *Information and Software Technology, 139,*.

Feng, S., Keung, J., Yu, X., Xiao, Y., Bennin, K. E., Kabir, M. A., & Zhang, M. (2021). Coste: Complexity-based oversampling technique to alleviate the class imbalance problem in software defect prediction. *Information and Software Technology, 129*, 106432.

García, V., Sánchez, J., & Mollineda, R. (2012). On the effectiveness of preprocessing methods when dealing with different levels of class imbalance. *Knowledge-Based Systems, 25,* 13–21.

Goel, L., Sharma, M., Khatri, S. K., & Damodaran, D. (2021). Cross-project defect prediction using data sampling for class imbalance learning: an empirical study. *International Journal of Parallel, Emergent and Distributed Systems, 36*(2), 130–143.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering, 26*, 653–661.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31,* 897–910.

Han, H., Wang, W.-Y., & Mao,B.-H. (2005). Borderline-smote: a new over-sampling method in imbalanced data sets learning. *In International conference on intelligent computing, pp.878–887*, Springer.

Hanley, J. A., & McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology, 143*, 29–36.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes in 2009. *IEEE 31st International Conference on Software Engineering*, pp.78–88.

Hassan, A. E., & Holt, R. C. (2005). The top ten list: dynamic fault prediction in 21st IEEE. *International Conference on Software Maintenance (ICSM'05)*, pp.263–272.

Haixiang, G., Yijing, L., Shang, J., Mingyun, G., Yuanyue, H., & Bing, G. (2017). Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications, 73,* 220–239.

Henderi, H., Wahyuningsih, T., & Rahwanto, E. (2021). Comparison of min-max normalization and z-score normalization in the k-nearest neighbor (knn) algorithm to test the accuracy of types of breast cancer. *International Journal of Informatics and Information Systems, 4*(1), 13–20.

Hosseini, S., Turhan, B., & Gunarathna, D. (2019). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering, 45,* 111–147.

Hosseini, S., Turhan, B., & Mäntylä, M. (2018). A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology, 95*, 296–312.

Jain, Y. K., & Bhandare, S. K. (2011). Min max normalization based data perturbation method for privacy protection. *International Journal of Computer & Communication Technology, 2*(8), 45–50.

Jiang, Y., Cukic, B., & Ma, Y. (2008). Techniques for evaluating fault prediction models. *Empirical Software Engineering, 13*, 561–595.

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K. I., Adams, B., & Hassan, A. E. (2010). Revisiting common bug prediction findings using effort-aware models in 2010. *IEEE International Conference on Software Maintenance*, pp.1–10

Khoshgoftaar, T., Allen, E., Goel, N., Nandi, A., & McMullan, J. (1996). Detection of software modules with high debug code churn in a very large legacy system. *In Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering,* pp.364–371.

Khoshgoftaar, T. M., & Allen, E. B. (2003). Ordering fault-prone software modules. *Software Quality Journal, 11*(1), 19–37.

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

Li, Y., Huang, Z., Wang, Y., & Fang, B. (2017). Evaluating data filter on cross-project defect prediction: Comparison and improvements. *IEEE Access, 5*, 25646–25656.

Limsettho, N., Bennin, K. E., Keung, J. W., Hata, H., & Matsumoto, K. (2018). Cross project defect prediction using class distribution estimation and oversampling. *Information and Software Technology, 100*, 87–102.

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

Malhotra, R., & Jain, J. (2022). Predicting defects in imbalanced data using resampling methods: an empirical investigation. *PeerJ Computer Science, 8*, e573.

Menzies, T., Dekhtyar, A., Distefano, J., & Greenwald, J. (2007). Problems with Precision: A Response to comments on'data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(9), 637–640.

Menzies, T., Jalali, O., Hihn, J., Baker, D., & Lum, K. (2010). Stable rankings for different effort models. *Automated Software Engineering, 17,* 409–437.

Mende, T., & Koschke, R. (2008). Revisiting the Evaluation of Defect Prediction Models in Proceedings of the 5th International Conference on Predictor Models in Software EngineeringPROMISE '09, (New York, NY, USA), pp.7:1—-7:10, ACM

Mende, T., Koschke, R., & Leszak, M. (2009). Evaluating defect prediction models for a large evolving software system in 2009. *13th European Conference on Software Maintenance and Reengineering, IEEE*

Mende, T., & Koschke, R. (2010). Effort-Aware Defect Prediction Models in 2010 14th.*European Conference on Software Maintenance and Reengineering*, pp.107–116

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering, 17*(4), 375–407.

Menardi, G., & Torelli, N. (2012). Training and assessing classification rules with imbalanced data. *Data Mining and Knowledge Discovery, 28,* 92–122.

Mnkandla, E., & Mpofu, B. (2016). Software defect prediction using process metrics elasticsearch engine case study in 2016 *International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pp.254–260

Moser Pedrycz, W., & Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *In Proceedings of the 30th international conference on Software engineering,* pp.181–190, ACM

Nagappan, N., & Ball, T. (2005). Static analysis tools as early indicators of pre-release defect density in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp.580–586.

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. *In Proceedings. 27th International Conference on Software Engineering ICSE 2005*, pp.284–292.

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. *In Proceedings of the 28th international conference on Software engineering*, pp.452–461, ACM

Ohlsson, N., & Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering, 22*(12), 886–894.

Ostrand, T., Weyuker, E., & Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering, 31*, 340–355.

Patro, S., & Sahu, K. K. (2015). Normalization: A preprocessing stage. arXiv preprint http://arxiv.org/abs/1503.06462.

Peters, F., Menzies, T., & Marcus, A. (2013). Better cross company defect prediction. *In Proceedings of the 10th Working Conference on Mining Software Repositories*, pp.409–418, IEEE Press.

Pudil, P., Novovicová, J., & Kittler, J. (1994). Floating search methods in feature selection. Pattern Recognition Letters, vol.15, no.11, pp.1119–1125

Qiu, S., Xu, H., Deng, J., Jiang, S., & Lu, L. (2019). Transfer Convolutional Neural Network for Cross-Project Defect Prediction. *Applied Sciences, 9*(13), 2660.

Rahman, F., & Devanbu, P. (2013). How, and why, process metrics are better in 2013 35th *International Conference on Software Engineering (ICSE)*, pp.432–441

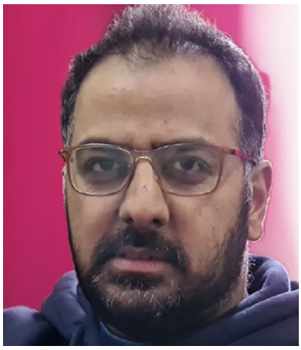
Ryu, D., Jang, J.-I., & Baik, J. (2017). A transfer cost-sensitive boosting approach for cross-project defect prediction. *Software Quality Journal, 25*(1), 235–272.

Suhag, V., Garg, A., Dubey, S. K., & Sharma, B. K. (2020). Analytical approach to cross project defect prediction. *In Soft Computing: Theories and Applications (M.Pant, T.K. Sharma, O.P. Verma, R.Singla, and A.Sikander, eds.)*, (Singapore), pp.713–736, Springer Singapore

Sun, Z., Li, J., Sun, H., & He, L. (2021). Cfps: Collaborative filtering based source projects selection for cross-project defect prediction. *Applied Soft Computing, 99*, 106940.

Tomar, D., & Agarwal, S. (2015). An effective weighted multi-class least squares twin support vector machine for imbalanced data classification. *International Journal of Computational Intelligence Systems, 8*(4), 761.
Tomar, D., & Agarwal, S. (2016). Prediction of defective software modules using class imbalance learning. *Applied Computational Intelligence and Soft Computing, 2016*, 1–12.
Turhan, B., Menzies, T., Bener, A. B., & Di Stefano, J. (2009). On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering, 14*(5), 540–578.
Turhan, B. (2012). On the dataset shift problem in software engineering. *Empirical Software Engineering, 17*(1–2), 62–74.
Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability, 62*(2), 434–443.
Xu, Z., Pang, S., Zhang, T., Luo,X.-P., Liu, J., Tang,Y.-T., Yu, X., Xue, L. (2019). Cross project defect prediction via balanced distribution adaptation based transfer learning. *Journal of Computer Science and Technology*, vol.34, pp.1039–1062.
Yu, Q., Qian, J., Jiang, S., Wu, Z., & Zhang, G. (2019). An empirical study on the effectiveness of feature selection for cross-project defect prediction. *IEEE Access, 7*, 35710–35718.
Zhang, H., & Zhang, X. (2007). Comments on data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, vol.33, pp.635–637
Zhao, Y. (2012). Rand data mining: Examples and case studies. Academic Press, 2012.
Zhou, Z.-H., & Liu, X.-Y. (2006). Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering, 18,* 63–77.
Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting Defects for Eclipse in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on, p.9*
Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting Faults from Cached History in *29th International Conference on Software Engineering (ICSE'07)*, pp.489–498
Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. *In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE 09, (New York, NY, USA), pp.91–100, ACM.

Nayeem Ahmad Bhat currently serves as a Lecturer in the Department of Computer Sciences at North Campus University of Kashmir. He earned his Ph.D. in Computer Sciences from the University of Kashmir, where he specialized in the area of software analytics. His research interests include Software defect prediction, AI in automated software testing, Machine learning. He has made significant contributions to the field of software defect prediction through his publications in top-tier journals and conferences. He is an active member of the research community. He serves as a reviewer for several prestigious journals and conferences.



Sheikh Umar Farooq is a Sr. Assistant Professor in the Department of Computer Sciences at North Campus of University of Kashmir. His research interests include empirical software testing, AI for automated testing, defect prediction, and Software Engineering Education. He did his PhD in Computer Sciences from the University of Kashmir. He has received the Basic Scientific research start-up grant from University Grants Commission in 2015 and research grant from ST&IC, DST in 2021. He is a professional member of ACM, CSI, ACM SIGSE, IACSIT SES, IAENG. Besides serving as reviewer for many prestigious journals, he has also served as PC member of many international conferences and workshops. He was General Chair of Conference on Software Engineering and Data Sciences (CoSEDS 2018) and also serves as co-organizer of Workshop on Emerging Software Engineering Education.