JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

MACHINE LEARNING-BASED PREDICTION OF SOFTWARE DEFECTS USING SOFTWARE METRICS

[1]Jahnavi Tirumala Sitty,
[1]GITAM university (B.Tech Electronics and communication),
[1]Visakhapatnam, Andhrapradesh, India.
[1]jahnavitirumala.2@gmail.com

Abstract : Software defect prediction ranks modules according to the fault risks faced by the module, to ensure limited time is spent in the maximum risk parts of the software. This paper conducts a comparison of the performance of nine machine learning classifiers (logistic regression, naive Bayes, k-NN, SVM, decision tree, random forest, histogram gradient boosting, XGBoost and a multilayer perceptron), which are trained with static code metrics for the prediction of defective modules. Experiments are conducted on 8 de-duplicated NASA MDP data sets, using each with 2 sets of 5-fold cross-validation (with and without SMOTE to train the folds only), and report accuracy, precision, recall, F1, AUC and MCC (1,440 model fits). Logistic regression, XGBoost and random forest have the highest mean AUC (0.790, 0.786 and 0.784) and are not significantly different under a Friedman-Nemenyi analysis, and are also not particularly high in terms of recall (not over 0.6 for the most part). After applying SMOTE to the data, each of the models improves recall by 0.311, F1 by 0.140, MCC by 0.084 and reduces precision by 0.058 and accuracy for each model, and afterwards, XGBoost achieves highest AUC at 0.793, highest MCC at 0.330 while lowering accuracy at 0.942. The sizes and Halstead measures are found to contain the majority of the signal using random-forest importance. According to the results, there is no single best classifier for every dataset, and certain evaluation metrics should be used with different resampling. Choice of an evaluation metric should be based on the relative cost of missed defects and false alarms, and resampling should be done accordingly.

IndexTerms - software defect prediction; software metrics; machine learning; class imbalance; SMOTE; NASA MDP; ensemble learning

I. INTRODUCTION

Faults that make it into production are expensive, and testing is always a limited amount and cannot be dedicated to every module. To solve these problems, one solution is to use a software defect prediction process (SDP), which is a module-ranking technology designed to flag the modules containing the highest number of defects. One way to get around these problems is by using a software defect prediction process (SDP), a module-ranking technology designed to identify top-risk modules to review and test first. In the last ten years, machine learning (ML) is the most popular method to address this ranker task. In a mapping study of 742 primary studies, the size of the literature is highlighted, and it is also indicated that the applicability of the studies to the business is limited [1]. The third study of the field gives similar results: There is strong evidence that ML-based techniques outperform traditional statistical methods on the datasets, but no single method is optimal [2].

The software metrics are usually fed into most SDP models. Static code metrics, including the McCabe cyclomatic complexity, the number of Halstead measures, and lines of code, are easy to gather, are in principle language-independent, and are publicly available. Approximately two-thirds of an extensive review of 102 studies on deep learning applied software metrics as input to their model [3]. As a result, metrics still play an essential part in addition to the supervised, semi-supervised, unsupervised and deep learning models discussed in the literature [4].

Predicting with metric techniques is more difficult than it sounds because there are three problems to overcome. The first is that defect information is very sparse and skewed toward "defective" modules. Second, labels and datasets are noisy: in a hand-coded data validation of SZZ data only approximately half of the commits identified as bug fixes actually were bug fixes [5]. Third, there is a lack of comparability of the reported results due to different datasets, different preprocessing techniques, different validation and/or different metrics used. Two large, resampling studies confirm that conclusions can vary with the balancing method used alone [6].

In this paper we provide a reproducible, controlled experiment with nine ML classifiers used for module-level defect prediction based on static code metrics. Its contributions are:

- A leakage-free evaluation protocol on eight NASA Metrics Data Program (MDP) datasets, with duplicate removal and all preprocessing and oversampling fitted inside the training folds only.
- A comparison of nine classifiers, with and without SMOTE oversampling, using six metrics: accuracy, precision, recall, F1-score, AUC and Matthews correlation coefficient (MCC).
- Statistical analysis with a Friedman test, Nemenyi critical difference and Wilcoxon signed-rank tests.
- An analysis of which static metrics the best model relies on, and of what oversampling does to each performance measure.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the datasets, preprocessing, models and evaluation protocol. Section 4 reports results. Section 5 discusses findings and threats to validity, and Section 6 concludes.

II. Related Work

2.1 Surveys and reviews

There are several recent reviews which map the field. This survey discusses open questions and trends as well as a survey of deep learning for defect prediction where the learning of semantic and structural code features is discussed [7]. A systematic review on learning-based SDP looks at the application scenarios, datasets, representation of source-code, prediction granularity, evaluation metrics and validation methods [8]. The algorithms that are mostly used in primary studies are CNN, LSTM and BiLSTM, while measuring the algorithm using a confusion matrix, F1, recall, precision, accuracy and AUC is most often used [9,10]. These reviews agree that it is important that a standardized mechanism for assessment exist, which underpins the motivation for this protocol.

2.2 Class imbalance and data quality

Resampling is the most common remedy for imbalance. In 2010, Bennin et al. applied eight resampling methods by a nearest-neighbor filter to 34 of the PROMISE datasets, using six prediction models, and obtained significant improvement in recall and g-measure but not in precision and false alarm rate [10]. Odejide et al. trained a decision tree (DT) and a random forest (RF) on four datasets using SMOTE, ADASYN, random oversampling (RO), random under sampling (RU) and Near Miss algorithms and concluded that random oversampling (RO) performs best [11]. A second challenge is label noise: a combination of under sampling, propensity-score-matching oversampling and noise reduction provided better data quality compared to 12 proposed methods, most notably in 11 PROMISE over 24 versions in the case of the highest noise level [12]. Overall, the study done in [6] concluded that the oversampling technique was generally superior to the under sampling technique for Apache data.

2.3 Ensembles, feature selection and explainability

Tree-based ensembles are good baselines for tabular metric data. Alazba and Aljamaan fine-tuned seven ensembles based on tree classifiers on 21 public datasets and demonstrated that their stacking model beats any single tuned tree-based model [13]. Tao et.al., evaluated 13 classifiers in combination with 10 sampling methods on ESEM2016 database with accuracy, AUC, F-measure and running time [14]. Yimer and Abrahim used seven ensembles and combined a correlation-based filter with all these ensembles on five AEEEM projects and found that the filter using Extra Trees worked best on all five [15]. Expect increasing expectation of explainability. Begum et al. experimented with 4 NASA datasets using the SMOTE algorithm with 13 models and LIME and SHAP, which focused on the metrics nosi, dit and cbo [16]. Finally, Haldar and Capretz applied SVM, k-NN, random forest and ANN on the PROMISE data using static code and effort data that present consistent results on independent and cross-project data [17]. They previously used SHAP for forecast information, specifically for the number of defects at the project level from cross-company metrics [18]

2.4 Deep, just-in-time, cross-project and LLM-based prediction

Deep models are extensions to the metric-based prediction. In the abstract [19] for a hybrid of CNN and Bi-LSTM on seven PROMISE java projects, it was mentioned that it achieved an average F-measure 25% better than the best baseline and an average AUC 18% better than the best baseline. Another CNN-MLP with a combination of abstract syntax tree (AST) features plus traditional metrics achieved an average F1 score of 0.703 compared to 0.668 for the next best method [20]. The level of change: A temporal convolutional network (TCN) with hierarchical attention was applied to product and process metrics for commits of 6 systems containing Java code to achieve accu 82.08% obtain using interpretability techniques that selects the most relevant metrics at the change level [22]. On cross project settings, the AUC and F1 obtained by Pearson feature selection with metric compensation transfer learning were improved by [23] and the F1 obtained by domain adaptation LSTM and SMOTE-ENN were improved by 1.48% to 33.03% on AEEEM, 25.10% to 42.60% on PROMISE [24]. Recently, there has been a review of 72 studies which categorizes the defect detection into dynamic and static groups, as well as the use of prompting and fine-tuning [25].

2.5 Gap

There is no direct comparison between numbers of studies listed above as these vary for different studies in terms of dataset, re-sample placement, validation and metrics. In this paper we propose only one protocol, applied to one class of datasets, that is able to measure the classical and ensemble learners, with and without oversampling, and test the differences statistically.

III. METHODOLOGY

Figure 1 summarizes the pipeline. Every step that learns from data (imputation, scaling, oversampling, model fitting) is fitted on the training fold only and then applied to the test fold.



Fig. 1. Experimental workflow. SMOTE (orange) is the only step toggled between the two experimental conditions.

3.1 Datasets

All eight of NASA's Metrics Data Program (MDP) datasets were used here and are available from a public version of the original database on GitHub. The rows correspond to the software module (function or method), and are accompanied by both static code metrics and binary labels for whether the software module is defective or not. The metrics introduced are McCabe complexity, Halstead metrics, line count and branch count. An exploration of the raw files will reveal many rows with exact duplicates, this may put almost identical models in the train and test folds in the folds and lead to overestimation of the score. We thus deduplicated our data (we removed duplicated rows (same metrics and label) and constant features). PC5 reduces to its lowest number: 17,186 rows to 1,830 unique modules. The cleaned data is detailed in table 1 and figure 2.

Table 1. Characteristics of the eight NASA MDP datasets before and after cleaning.

Dataset	Raw modules	Modules after cleaning	Features	Defective	Defect rate (%)
CM1	505	456	37	47	10.31
JM1	10,878	8,905	21	2,003	22.49
KC1	2,107	1,210	21	314	25.95
MW1	403	382	37	31	8.12
PC1	1,107	961	37	69	7.18
PC3	1,563	1,439	37	153	10.63
PC4	1,458	1,344	37	177	13.17
PC5	17,186	1,830	38	484	26.45

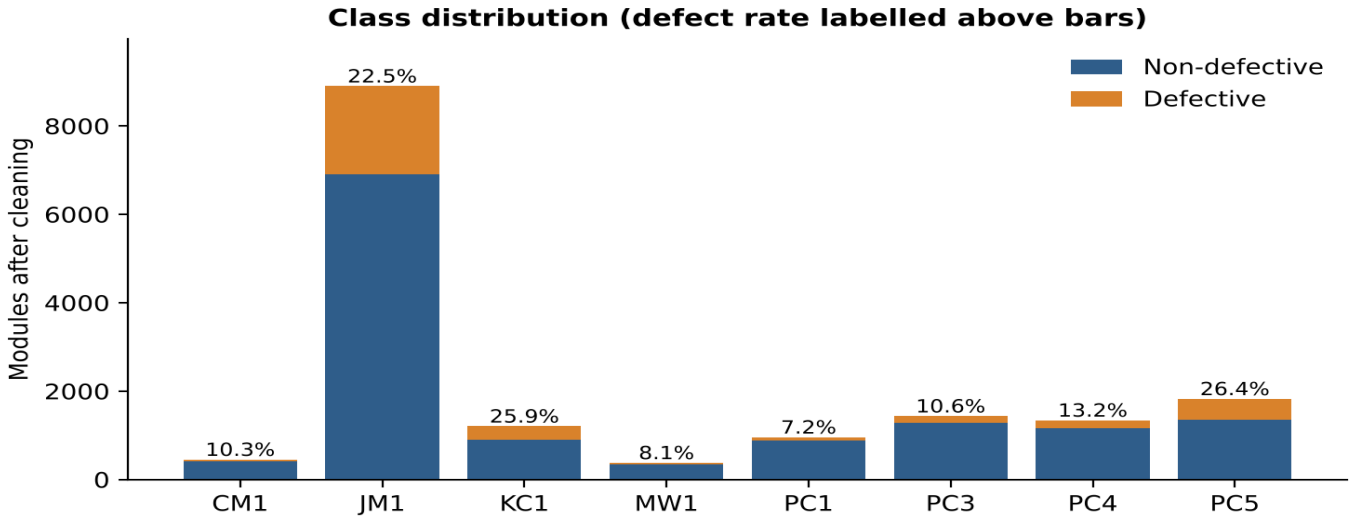

Fig. 2. Class distribution after cleaning. Defect rates range from 7.2% to 26.4%.

Many metrics are strongly related, because most are functions of code size. Figure 3 shows the Spearman correlation of selected JM1 metrics. The Halstead measures and executable lines are almost perfectly rank-correlated, and total lines of code correlate with cyclomatic complexity at 0.76. This redundancy matters for models that are sensitive to collinearity.

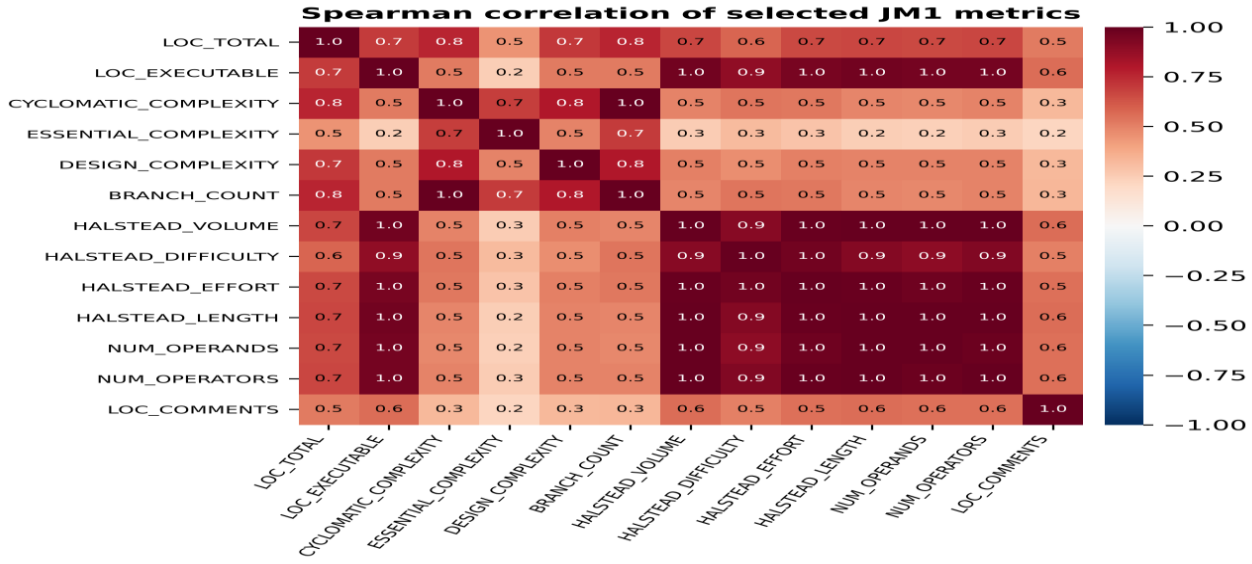

Fig. 3. Spearman correlation of 13 selected JM1 metrics.

3.2 Preprocessing

For every fold of training, the number of missing values are replaced by training median. We apply skewness remediation log(1 + x) and then z -score using the training statistics, where metrics are heavily skewed to the right. The distance-based models and the linear models are helped by the transform while the tree-based ones are not affected.

3.3 Class imbalance

There are two states being compared. In the baseline condition the training fold is used as it is. For SMOTE condition minority class data is oversampled at 1:1 ratio using SMOTE with k=5 neighbours (which reduces to fewer when the folds have fewer minority samples). The reported precision and recall are true estimates of the class distribution (the test fold won't be re-sampled).

3.4 Classifiers

Nine of the most important model families are represented by classifiers (Table 2). The choice of hyperparameters is unsittable, but limited to sensible values and not to used for tuning; hence the difference reflect the algorithm and not on the tuning budget.

Table 2. Classifiers and their settings. All use random seed 42 where applicable.

Code	Classifier	Main settings
LR	Logistic regression	L2 penalty, up to 2,000 iterations
NB	Gaussian naive Bayes	Default
KNN	k-nearest neighbours	k = 7
SVM	Support vector machine	RBF kernel; decision-function scores for AUC
DT	Decision tree	Maximum depth 8
RF	Random forest	300 trees
HGB	Histogram gradient boosting	Default
XGB	XGBoost	300 trees, depth 4, learning rate 0.05, subsample 0.8, column sample 0.8
MLP	Multilayer perceptron	Layers (64, 32), early stopping, up to 400 iterations

3.5 Evaluation protocol and metrics

Each dataset is evaluated with 5-fold stratified cross-validation repeated twice (10 test folds). Results are averaged over folds within each dataset first, and then over the eight datasets, so large datasets do not dominate. With nine classifiers, two conditions and eight datasets, the study comprises 1,440 model fits. We report accuracy, precision, recall, F1-score, AUC and MCC:

This is only provided for completeness since a classifier that classifies every object as "clean" will achieve accuracy of approximately 74%-93% on these data. Bigger AUC indicates the higher quality of the rankings regardless of the threshold and MCC is robust to imbalance.

3.6 Statistical analysis

The Friedman test at a level of significance (α) = 0.05 is performed for mean ranks across the eight datasets followed by the Nemenyi critical difference (CD) test at level (α) = 0.05. The Wilcoxon signed-rank test is used to evaluate the effect of SMOTE over the 72 pairs of classifiers and datasets, respectively, over the eight datasets to test each one. Pooled test assumes that pairs are independent, although they are not; this is a descriptive test, not a descriptive inference.

IV. RESULTS

Gradient-boosted and random-forest models work best on AUC and SMOTE increases the recall for nearly all classifiers whilst decreasing precision and accuracy. The following are all averages over the 8 data sets and each a mean of 10 test folds.

4.1 Baseline performance (no resampling)

If not the resampling accuracy is 0.85 for most models and low accuracy for recall (Table 3). The SVM and MLP are able to identify only 9.6% and 11.3% of the defective modules. Logistic regression outperforms other models, with mean AUC of 0.790, while XGBoost and random forest models come close with mean AUC 0.786 and 0.784 respectively. The highest precision is obtained by Naive Bayes (0.285) and the highest recall is achieved by Naive Bayes (0.636) while the highest f1 score is achieved by Naive Bayes (0.378). The best MCC (0.287) is obtained using the histogram gradient boosting.

Table 3. Mean performance without resampling (average over 8 datasets).

Model	Accuracy	Precision	Recall	F1	AUC	MCC
LR	0.854	0.533	0.182	0.256	0.790	0.241
NB	0.707	0.285	0.636	0.378	0.748	0.260
KNN	0.845	0.434	0.207	0.268	0.740	0.215
SVM	0.855	0.353	0.096	0.147	0.732	0.142
DT	0.821	0.366	0.263	0.295	0.638	0.205
RF	0.850	0.447	0.215	0.284	0.784	0.233
HGB	0.850	0.498	0.279	0.346	0.777	0.287
XGB	0.852	0.497	0.246	0.317	0.786	0.270
MLP	0.844	0.322	0.113	0.158	0.627	0.125

Figure 4 shows that the best model depends on the dataset. Logistic regression is best on CM1 (0.813), PC1 (0.854) and PC3 (0.834). XGBoost is best on JM1 (0.702), KC1 (0.686) and PC4 (0.939), and random forest on MW1 (0.786) and PC5 (0.780). The MLP is near chance on CM1, MW1 and PC1 (AUC 0.47 to 0.52). A likely cause is that its early-stopping validation split holds very few defective modules in these small, imbalanced sets, but we did not test this explanation.

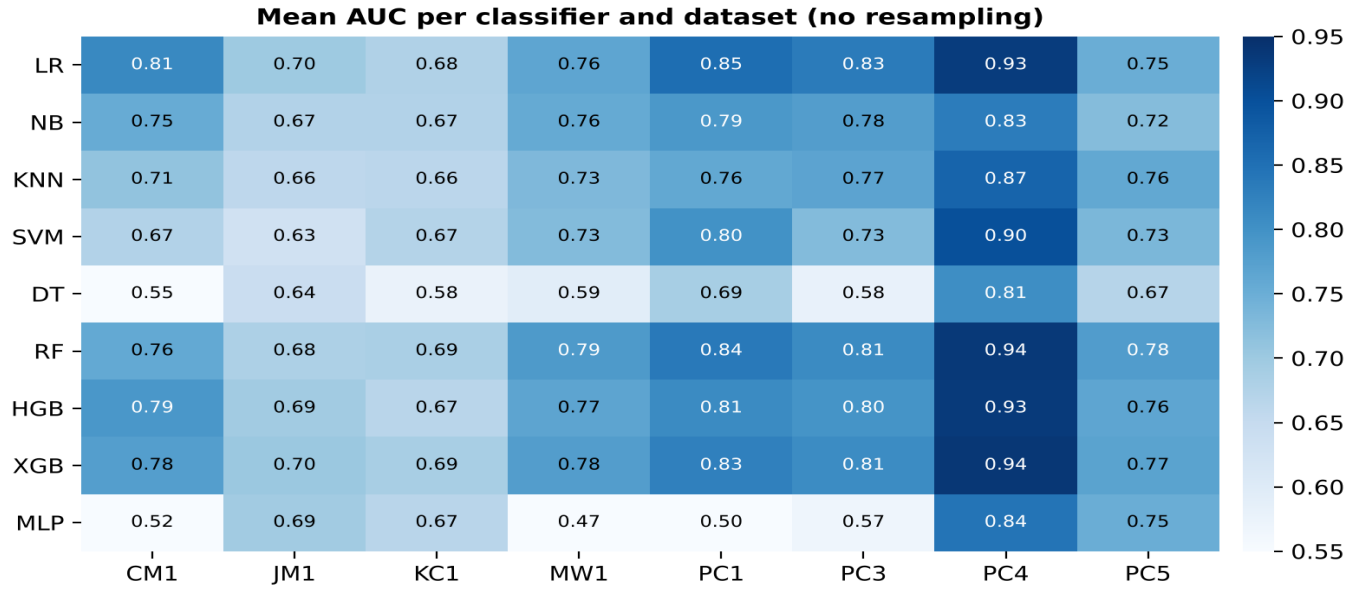

Fig. 4. Mean AUC for each classifier and dataset without resampling.

The datasets differ strongly in difficulty. PC4 is the easiest (best AUC 0.939) and JM1 and KC1 are the hardest (best AUC about 0.70). Figure 5 shows out-of-fold ROC curves on JM1, where XGBoost (0.704) and logistic regression (0.700) are almost indistinguishable and the SVM is lowest (0.633).

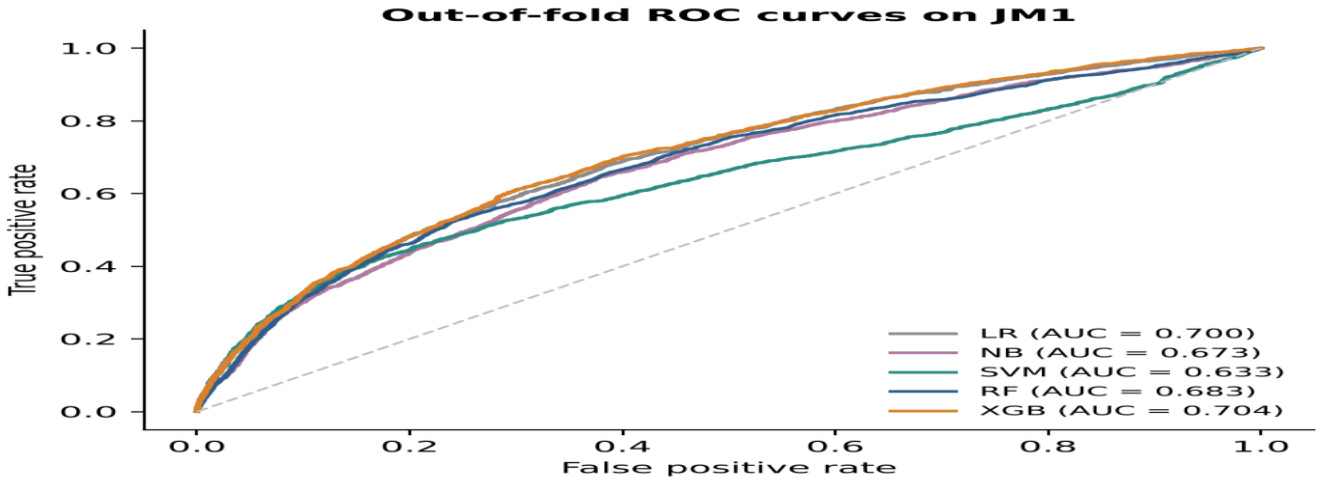

Fig. 5. Out-of-fold ROC curves on JM1 for five classifiers.

4.2 Statistical comparison of classifiers

The Friedman test finds significant differences between classifiers on AUC ($\chi^2 = 48.03$, $p < 0.001$), F1 ($\chi^2 = 44.20$, $p < 0.001$) and MCC ($\chi^2 = 32.07$, $p < 0.001$). Table 4 gives mean ranks, where lower is better. XGBoost has the best average AUC rank (2.00) and the random forest and logistic regression follow (2.38, 2.62). The Nemenyi critical difference for 9 classifiers and 8 datasets is 4.25. By this criterion XGBoost is significantly better than the decision tree, MLP and SVM on AUC; its difference from k-NN (4.25) sits exactly at the threshold. The top four models (XGB, RF, LR, HGB) are not significantly different from one another. On F1 and MCC the ranking changes: naive Bayes (F1 rank 1.75) and histogram gradient boosting (F1 rank 2.38, MCC rank 2.12) lead.

Table 4. Mean ranks across 8 datasets without resampling (lower is better; Nemenyi CD = 4.25)

Model	AUC rank	F1 rank	MCC rank
LR	2.62	6.38	4.62
NB	5.88	1.75	4.00
KNN	6.25	4.50	5.25
SVM	6.50	8.50	7.62
DT	8.38	4.75	6.50
RF	2.38	5.00	4.62
HGB	3.62	2.38	2.12
XGB	2.00	3.88	2.75
MLP	7.38	7.88	7.50

4.3 Effect of SMOTE

With SMOTE, recall rises sharply and the F1 ranking changes (Table 5). Mean recall for the SVM goes from 0.096 to 0.640 and for the MLP from 0.113 to 0.608. XGBoost has the best AUC (0.793) and MCC (0.330), while the MLP has the best F1 (0.441), followed by the SVM (0.436) and logistic regression (0.433). Accuracy falls for every model, from 0.012 for HGB to 0.146 for k-NN.

Table 5. Mean performance with SMOTE applied to training folds only (average over 8 datasets).

Model	Accuracy	Precision	Recall	F1	AUC	MCC
LR	0.742	0.326	0.686	0.433	0.785	0.325
NB	0.684	0.276	0.687	0.379	0.751	0.263
KNN	0.699	0.287	0.690	0.395	0.752	0.278
SVM	0.764	0.339	0.640	0.436	0.783	0.327
DT	0.763	0.306	0.497	0.372	0.670	0.244
RF	0.829	0.421	0.421	0.416	0.785	0.315
HGB	0.838	0.453	0.396	0.416	0.791	0.323
XGB	0.837	0.453	0.412	0.423	0.793	0.330
MLP	0.773	0.356	0.608	0.441	0.785	0.327

Figure 6 and Table 6 isolate the change. Pooled over all 72 classifier-dataset pairs, SMOTE changes recall by +0.311, precision by −0.058, F1 by +0.140, AUC by +0.030 and MCC by +0.084 (Wilcoxon $p < 0.001$ for each). The gain is largest for models that were otherwise biased toward the majority class (SVM, MLP, LR, k-NN), and smallest for naive Bayes and the tree ensembles. The MLP gains 0.158 in AUC, which suggests that balanced training data also stabilizes its training. For RF, XGB, k-NN and NB the AUC change is not significant per model.

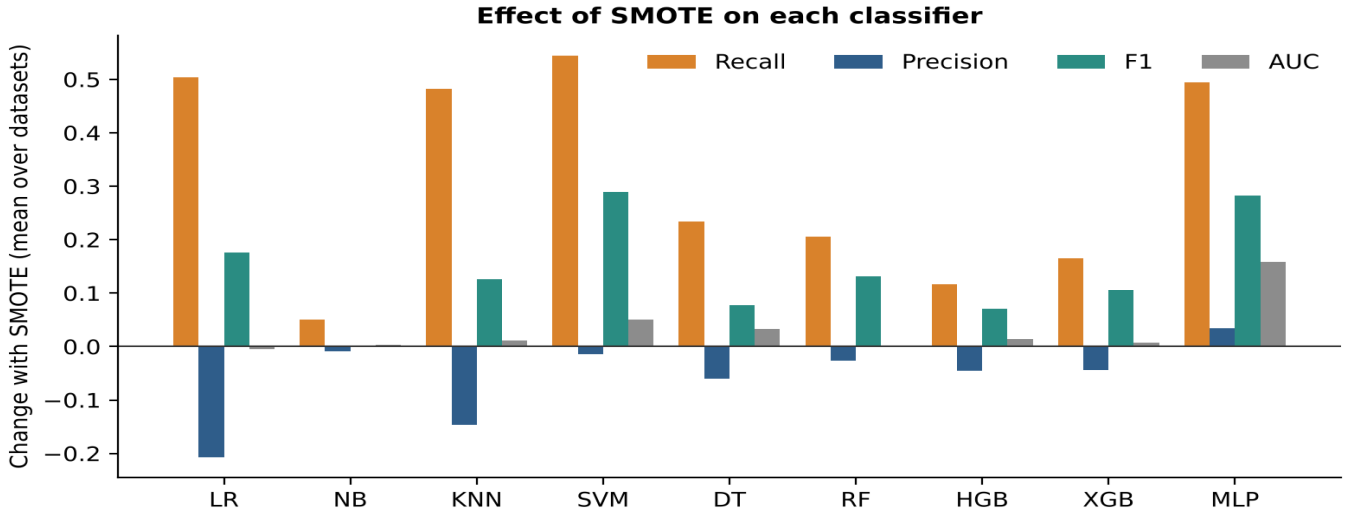

Fig. 6. Change in recall, precision, F1 and AUC when SMOTE is added (mean over 8 datasets).

Table 6. Change caused by SMOTE (SMOTE minus baseline) and per-model Wilcoxon p-values over 8 datasets (two-sided; the smallest attainable p is 0.0078). No multiple-comparison correction was applied.

Model	Δ Recall	Δ Precision	Δ F1	Δ AUC	Δ MCC	p (F1)	p (AUC)
LR	+0.504	−0.207	+0.177	−0.005	+0.084	0.008	0.023
NB	+0.051	−0.009	+0.001	+0.003	+0.003	0.844	0.078
KNN	+0.483	−0.147	+0.126	+0.012	+0.063	0.016	0.148
SVM	+0.545	−0.014	+0.289	+0.051	+0.185	0.008	0.008
DT	+0.234	−0.060	+0.077	+0.032	+0.039	0.016	0.039
RF	+0.206	−0.026	+0.131	+0.001	+0.081	0.008	0.641
HGB	+0.117	−0.045	+0.070	+0.014	+0.036	0.039	0.016
XGB	+0.165	−0.044	+0.105	+0.007	+0.060	0.008	0.148
MLP	+0.495	+0.034	+0.283	+0.158	+0.202	0.008	0.039

.

With eight datasets, the smallest two-sided p-value is 0.0078, so no per-model test survives a Bonferroni correction across the 18 tests. The more reliable evidence is directional: recall increases on all eight datasets for every one of the nine classifiers ($p = 0.0078$ each).

4.4 Which metrics matter

We fitted a class-weighted random forest (500 trees) on the 21 metrics shared by all eight datasets and averaged impurity-based importance across datasets (Figure 7). Importance is spread thinly: the top metric, Halstead content, scores 0.077 against a uniform value of 0.048. Size-related measures dominate the top ten, namely blank lines (0.077), comment lines (0.065), total lines (0.064), code-and-comment lines (0.062) and executable lines (0.049), together with Halstead volume, unique operands, program time and effort. Complexity measures such as cyclomatic complexity do not appear in the top ten.

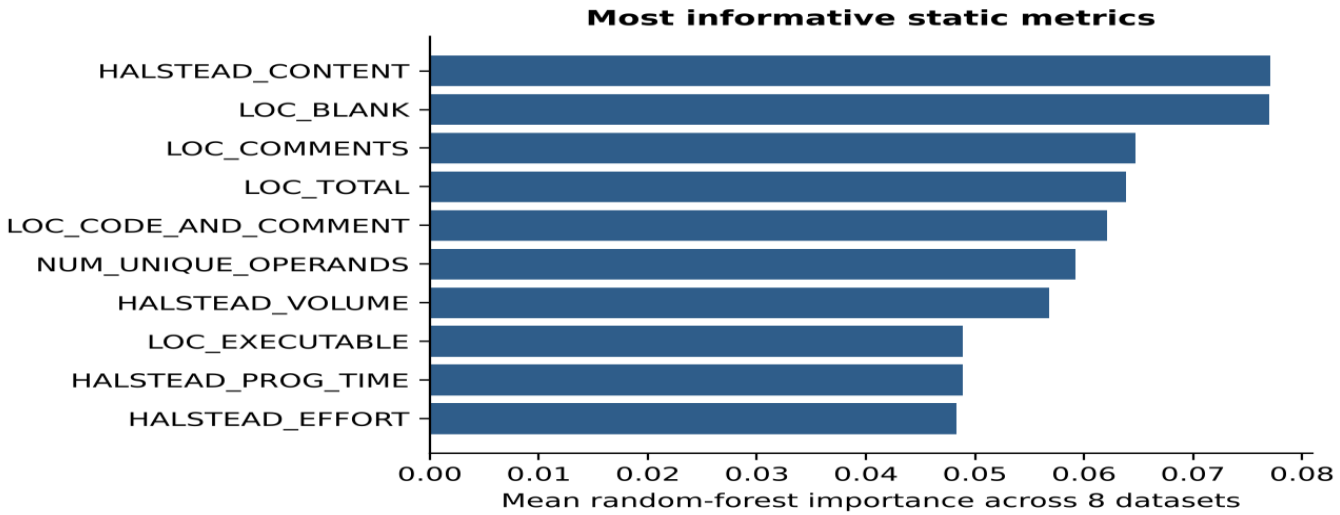

Fig. 7. Mean random-forest importance of the 10 most informative metrics across the eight datasets.

V. Discussion

5.1 Main findings

There is no single classifier that is always ahead of the others. There is no statistical difference between XGBoost and random forest or logistic regression for AUC; which are better will depend on the dataset. This is in line with both tertiary findings: 1) no single technique is best on all the datasets [2] and 2) no single dataset or model is best on all the datasets [4]. On small datasets like CM1 and PC1, an ensemble is competitive with a simple, well-scaled linear model; a reminder when resource limitations are tight when tuning budgets. Tree ensembles have a greater median rank across the datasets, and a lower variance, so they are the more secure choice.

The order of the metric changes the winner. This is where AUC is more in favor of ensembles and logistic regression, while F1 and MCC are more in favor of naive Bayes and gradient boosting. This is consistent with the multitude of evaluation metrics found in the literature [8, 9]: that reports that use only accuracy, or only AUC, can only lead to misunderstanding.

5.2 Oversampling

SMOTE sacrifices in terms of precision for recall. In our results it improves the recall for every dataset for every classifier it gets, and decreases precision in eight of nine cases. The same trend was observed in some earlier works reporting better recall and g-measure for resampling compared to undersampling on 34 PROMISE datasets [10] and with oversampling consistently outperforming undersampling [6, 11]. Use is dependent on the costs. For low review capacity, the higher precisions of the baseline ensembles will benefit; for a false alarm cost that is much smaller than the cost of a defect, the SMOTE condition would be preferred. Oversampling gains are greatest for the models with the lowest initial accuracy, and tree ensembles actually gain the least in AUC, making oversampling more like a shifting tool that sets rank thresholds than an accuracy boosting tool.

5.3 Comparison with published results

The absolute scores (best mean AUC 0.79 and F1 0.44) are much lesser compared to the exceptional improvements seen in the studies of deep and hybrids [19]-[20]-[24]. While comparisons of datasets are not recommended, direct comparisons of these results are not safe because of differences in the datasets and in the location of samples for validation and resampling. We are not asserting the errors in the results of these direct comparisons. We believe the following two things reduce the score in our protocol: we do not train the oversampling data outside of the data folds and there is some duplicate rows removed. Modeled by a name label, additional constraint on any metric model [5,12]. Accurate ensembles such as stacking and tuned ensembles gain accuracy beyond what our untuned ensembles can give [13] and [14, respectively] and deep/LLM-based approaches improve upon information that is missing from static metrics [3, 25, respectively].

5.4 Interpretation of metric importance

The importance analysis indicates that module size is the most important so that this factor is similar to the high degree of correlation that was obtained in Figure 3. SHAP and LIME have been used in this application, and their results indicated that static code metrics are predictors [16, 17]. The ordering of each metric should be interpreted with caution; the strong conclusion that size and Halstead measures play a greater role than control-flow complexity of the metric is more meaningful than the ranking.

5.5 Threats to validity

- **Internal validity.** Hyperparameters were fixed rather than tuned, which may disadvantage models that are sensitive to tuning (for example the MLP). Results use a single random seed for model initialization and two cross-validation repeats.
- **Construct validity.** Labels come from the original NASA datasets, whose quality has been questioned, and module-level labels ignore defect severity and count.
- **External validity.** The eight datasets are from NASA projects in a small number of languages and domains. Results may not carry over to modern open-source projects or to just-in-time prediction [21], [22], and no cross-project setting was tested [23].
- **Statistical validity.** With eight datasets, per-model tests have low power and no multiple-comparison correction was applied to the Wilcoxon tests; the Friedman and Nemenyi results are more reliable than individual p-values.

VI. Conclusion

In this study, nine machine learning classifiers were compared with each other in module-level defect prediction, on eight de duplicated NASA MDP datasets with and without SMOTE, in the leakage-free protocol. Three conclusions follow. Firstly, XGBoost, random forest and logistic regression can be used for prediction of quality (mean AUC 0.784 to 0.790) and the ranking among the three is dataset dependent. Second, as default, recall levels are low for most models and SMOTE increases mean recall by 0.311 and F1 by 0.140; Recall decreases by 0.058 and accuracy decreases by 0.011 for every model. Third, the size/halstead measures contain most of the information in the metrics.

In the practical aspects, it means employing tree ensembles or regularized linear models as a base and reporting only the AUC along with the F1 and MCC in addition and perform ensembling or oversampling only in the case that it is economically more advantageous due to the relative cost of missed defects and false alarms.

References

[1] S. Stradowski and L. Madeyski, "Machine learning in software defect prediction: A business-driven systematic mapping study," Inf. Softw. Technol., vol. 155, Art. no. 107128, 2023, doi: 10.1016/j.infsof.2022.107128.

[2] A. Taskeen, S. U. R. Khan, and E. A. Felix, "A research landscape on software defect prediction," J. Softw. Evol. Process, vol. 35, no. 12, 2023, doi: 10.1002/smr.2549.

[3] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," J. Syst. Softw., Art. no. 111537, 2023, doi: 10.1016/j.jss.2022.111537.

[4] Y. L. Zhang and N. Liu, "Investigation and research on several key issues of software defect prediction," IET Softw., vol. 2025, no. 1, 2025, doi: 10.1049/sfw2/6615496.

[5] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection," Empir. Softw. Eng., vol. 27, no. 2, Art. no. 42, 2022, doi: 10.1007/s10664-021-10092-4.

[6] R. Malhotra and J. Jain, "Predicting defects in imbalanced data using resampling methods: An empirical investigation," PeerJ Comput. Sci., vol. 8, Art. no. e573, 2022, doi: 10.7717/peerj-cs.573.

[7] E. N. Akimova et al., "A survey on software defect prediction using deep learning," Mathematics, vol. 9, no. 11, Art. no. 1180, 2021, doi: 10.3390/math9111180.

[8] C. Li, D. Li, H. Li, W. E. Wong, and M. Zhao, "A systematic review of learning-based software defect prediction," J. Internet Technol., vol. 26, no. 4, pp. 501–511, Jul. 2025.

[9] A. T. Hasan and S. M. Mohi-Aldeen, "Software defect prediction based on deep learning algorithms: A systematic literature review," Al-Rafidain J. Comput. Sci. Math., vol. 19, no. 1, pp. 67–79, Jun. 2025, doi: 10.33899/csmj.2025.156086.1160.

[10] K. E. Bennin, A. Tahir, S. G. MacDonell, and J. Börstler, "An empirical study on the effectiveness of data resampling approaches for cross-project software defect prediction," IET Softw., vol. 16, no. 2, pp. 185–199, 2022, doi: 10.1049/sfw2.12052.

[11] B. J. Odejide et al., "An empirical study on data sampling methods in addressing class imbalance problem in software defect prediction," in Computer Science On-line Conference (CSOC 2022), Lecture Notes in Networks and Systems, vol. 501. Cham, Switzerland: Springer, 2022, pp. 594–610, doi: 10.1007/978-3-031-09070-7_49.

[12] H. Shi, J. Ai, J. Liu, and J. Xu, "Improving software defect prediction in noisy imbalanced datasets," Appl. Sci., vol. 13, no. 18, Art. no. 10466, 2023, doi: 10.3390/app131810466.

[13] A. Alazba and H. Aljamaan, "Software defect prediction using stacking generalization of optimized tree-based ensembles," Appl. Sci., vol. 12, no. 9, Art. no. 4577, 2022, doi: 10.3390/app12094577.

[14] H. Tao et al., "A comparative study of software defect binomial classification prediction models based on machine learning," Softw. Qual. J., vol. 32, no. 3, pp. 1203–1237, Sep. 2024, doi: 10.1007/s11219-024-09683-3.

[15] B. S. Yimer and D. A. Abrahim, "Enhanced software defect prediction using ensemble learning with correlation-based feature selection and SMOTE," Ethiop. J. Sci. Sustain. Dev., vol. 13, no. 1, pp. 60–70, 2025, doi: 10.20372/ejssdastu:v13.i1.2026.1111.

[16] M. Begum, M. H. Shuvo, I. Ashraf, A. Al Mamun, J. Uddin, and M. A. Samad, "Software defects identification: Results using machine learning and explainable artificial intelligence techniques," IEEE Access, vol. 11, pp. 132750–132765, 2023, doi: 10.1109/ACCESS.2023.3329051.

[17] S. Haldar and L. F. Capretz, "Interpretable software defect prediction from project effort and static code metrics," Computers, vol. 13, no. 2, Art. no. 52, 2024, doi: 10.3390/computers13020052.

[18] S. Haldar and L. F. Capretz, "Explainable software defect prediction from cross company project metrics using machine learning," in Proc. 7th Int. Conf. Intell. Comput. Control Syst. (ICICCS), Madurai, India, 2023, pp. 150–157, doi: 10.1109/ICICCS56967.2023.10142534.

[19] A. B. Farid, E. M. Fathy, A. S. Eldin, and L. A. Abd-Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)," PeerJ Comput. Sci., vol. 7, Art. no. e739, 2021, doi: 10.7717/peerj-cs.739.

[20] A. Abdu et al., "Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model," Sci. Rep., vol. 14, Art. no. 14771, 2024, doi: 10.1038/s41598-024-65639-4.

[21] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Just-in-time software defect prediction using deep temporal convolutional networks," Neural Comput. Appl., vol. 34, no. 5, pp. 3981–4001, 2022, doi: 10.1007/s00521-021-06659-3.

[22] R. A. Dos Santos, "Just-in-time software defect prediction using a deep learning-based model," New Trends Comput. Sci., vol. 2, no. 2, pp. 91–100, 2024, doi: 10.3846/ntcs.2024.22274.

[23] J. Chen, X. Wang, S. Cai, J. Xu, J. Chen, and H. Chen, "A software defect prediction method with metric compensation based on feature selection and transfer learning," Front. Inf. Technol. Electron. Eng., vol. 23, no. 5, pp. 715–731, 2022, doi: 10.1631/FITEE.2100468.

[24] K. Javed, R. Shengbing, M. Asim, and M. A. Wani, "Cross-project defect prediction based on domain adaptation and LSTM optimization," Algorithms, vol. 17, no. 5, Art. no. 175, 2024, doi: 10.3390/a17050175.

[25] Y. Chen et al., "Software defect detection using large language models: A literature review," Front. Comput. Sci., vol. 20, no. 6, Art. no. 2006202, 2026, doi: 10.1007/s11704-025-40672-2.