



Empirical Evaluation of Machine Learning Classifiers for Static Metric Software Defect Prediction

Ishrat Afreen¹, Narendra Parmar², Vishal Shrivastava³, Gagan Sharma⁴

¹Research Scholar, ²Assistant Professor, ^{3/4}Associate Professor

^{1,2,3,4}Department of Computer Science and Engineering, RKDF University, Bhopal, India

¹ishratafreen129@gmail.com, ²narendrarkdfac.rkdf@gmail.com, ³vishal.rkdf@gmail.com,

⁴gagansharma.cs@gmail.com

ABSTRACT

With the exponential scaling and structural dependency of complex code deployment, maintaining high software quality parameters has surfaced as a primary bottleneck within the contemporary software development lifecycle. Late-stage bug debugging and software defects noticeably amplify overall implementation costs, consume vast development schedules, and exhaust institutional engineering hours. Consequently, early fault localization and predictive defect classification act as key economic levers, enabling optimal programmatic planning and allocation of limited validation resources. This paper provides a result-oriented comparative mapping of data mining schemes optimized specifically for empirical Software Defect Prediction (SDP) using supervised classification rules. We comprehensively detail the theoretical profiles of multiple paradigms: Bayesian modelling, rule-based induction engines, logistic formulations, and decision tree topologies. Utilizing statistical hold-out validation across twelve high-dimensional legacy systems extracted from the public NASA Metric Data Program (MDP) repository, we baseline the performance of eight prominent learners—Naive Bayes (NB), Logistic Regression (LOG), Decision Table (DT), OneR, PART, JRip, J48, and J48Graft. Evaluation across standard predictive dimensions (Accuracy, Sensitivity, Specificity, Balance, and Receiver Operating Characteristic (ROC) Area) proves that model predictive dominance varies considerably by system architectural traits. The findings confirm that optimizing a defect framework demands a modular, context-driven selection strategy rather than a static monolithic algorithm.

Keywords: Software Defect Prediction, Machine Learning Classifiers, NASA MDP Repository, Classification Rule Mining, Confusion Matrix, Generalization.

I. INTRODUCTION

The escalating demand for high-reliability software modules across mission-critical frameworks has shifted testing phases to the forefront of operational engineering concerns [1]. Empirically, software faults are not uniformly spread across system files but instead follow a Pareto distribution, accumulating selectively within a narrow cluster of structural modules [2]. Identifying these highly vulnerable sections prior to execution reduces overall structural validation costs, minimizes working effort, and balances time constraints [3].

Classical defect estimation methods heavily relied on continuous mathematical regression engines [4]. However, point estimation remains highly unpredictable during early project phases when internal metrics are incomplete [5]. Supervised machine learning transforms this

task into a binary classification problem, grouping software blocks into defective or non-defective classes based on code attribute behaviours [6].

While various machine learning algorithms are used globally for software fault categorization, choosing an optimal layout remains difficult due to noise and structural imbalances in legacy data streams [7]. This research introduces an empirical evaluation framework designed to test eight machine learning miners on twelve open-source datasets from the NASA Metric Data Program (MDP). By evaluating performance metrics through confusion matrices and area under the curve (AUC) bounds, we provide software quality managers with a practical, result-driven guide for selecting predictive analytics engines [8], [9].

II. RELATED WORK

A key focus in modern software engineering is analysing historical development data to predict potential failure trends [1]. For example, Bibi et al. introduced Regression via Classification (RvC), which addresses bug density by discretizing continuous targets into distinct intervals for classification, then reversing the output back into numeric forms [4].

A foundational standard for mining code repositories was set by Menzies, Greenwald, and Frank (MGF) [5]. They assessed the performance of Naive Bayes and Rule Induction models across 10 NASA MDP modules, highlighting the importance of static metrics. However, later critics noted that using the entire dataset simultaneously for attribute ranking introduces data leakage, distorting predictive reliability [9].

Alternative methodologies have emerged to address these issues. Gondra utilized Artificial Neural Networks (ANN) to map complex structural patterns [2], while Oral and Bener focused on localized hardware code by testing Multilayer Perceptrons, Naive Bayes, and Voting Feature Intervals across seven embedded software modules [6]. In terms of rule mining, Zhang and Shang applied Association Rule Classification via CBA2, showing its competitive advantages over standard decision trees like C4.5 [7]. Song et al. later introduced a joint defect-proneness prediction framework evaluated through cross-validations, confirming the benefits of thorough data validation over static pipelines [10].

III. PROPOSED EXPERIMENTAL FRAMEWORK

The experimental framework consists of two main processing components: Scheme Evaluation and Defect Prediction.

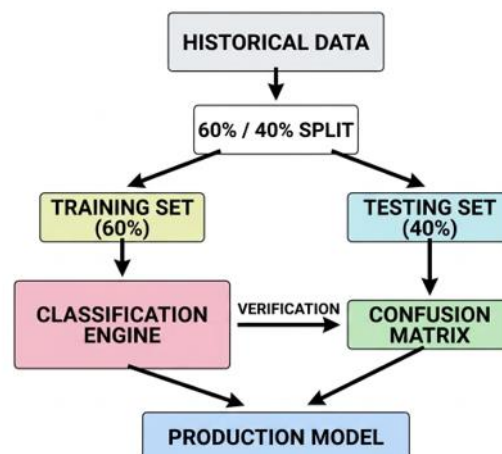


Fig.1 Machine Learning Scheme Evaluation Pipeline showing data splitting, model training, and performance metrics

A. Scheme Evaluation Stage

A common pitfall in empirical software engineering is selecting an operational algorithm without testing its behaviour on future data [9]. To prevent generalization errors, our framework separates the testing dataset completely from model creation [10]. Each dataset undergoes an automated 60% training and 40% testing split. To eliminate any ordering bias, the files are randomized over twelve separate iterations, producing a reliable set of evaluation matrices.

B. Defect Prediction Stage

Once the evaluation stage identifies the top-performing learning configurations, the entire historical dataset is used to build the final operational predictor [11]. This maximizing approach increases the model's generalization capabilities, preparing it to process and classify incoming, unlabelled code modules [12].

IV. CORE MINING ALGORITHMS EVALUATED

We evaluate eight machine learning classifiers spanning four algorithmic categories [13]:

A. Bayesian Classification

Naive Bayes (NB): Utilizes Bayes' Theorem under the assumption of strict conditional independence among predictors [5]. It calculates posterior probabilities using the formula:

$$P(c|x) = [P(x|c) \times P(c)] / P(x)$$

This approach simplifies parameter estimation, making it highly efficient for massive codebases [11].

B. Rule-Based Classification

1) Decision Table (DT): Constructs a descriptive mapping table that links specific metric combinations directly to a majority class label [3].

i. OneR: Generates a minimal, single-attribute rule system by selecting the individual feature with the lowest overall training error rate [13].

ii. JRip: Implements the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm, refining rules recursively to optimize classification accuracy [3].

iii. PART: Produces partial decision trees and converts their paths into direct IF-THEN production rules without requiring global optimization [13].

C. Logistic Modeling

Logistic Regression (LOG): Maps non-linear, multi-dimensional metric features into a continuous probability range bounded between 0 and 1 via the logistic sigmoid function [8]:

$$F(z) = 1 / (1 + e^{-z})$$

D. Decision Tree Formulations

1) J48: A Java-based implementation of the traditional Quinlan C4.5 algorithm, which splits tree nodes by maximizing information gain [12].

2) J48Graft: Extends J48 by grafting alternative sub-branches onto empty or underrepresented leaf locations, reducing model variance and improving accuracy on noisy datasets [12].

V. EXPERIMENTAL SETUP AND DATASET PROFILES

Simulations were executed using the Weka data mining suite (v3.6.9) [12]. The experimental evaluations relied on twelve open-source data repositories from the NASA Metric Data Program (MDP) [9]. These modules include static attributes like McCabe cyclomatic complexity, Halstead software metrics, and lines of code (LOC) counts.

TABLE I System Architectural Properties and Imbalance Matrix

ID	Core Domain	Lang	LOC	Mod	Def	Ratio
CM1	Spacecraft Inst.	C	17K	344	42	12.22%
JM1	Real-time Mission	C	315K	9593	1759	18.34%
KC1	Ground Storage	C++	43K	2096	325	15.50%
KC3	Ground Collection	Java	8K	200	36	18.00%
MC1	Video Flight Control	C	67K	9277	68	0.73%
MC2	Spacecraft Core	C	6K	127	44	34.65%
MW1	Core DB Interface	C	8K	264	27	10.23%
PC1	Earth Orbiting Flight	C	26K	759	61	8.04%
PC2	Orbit Data Proc.	C	26K	1585	16	1.00%
PC3	Attitude Control Ground	C	36K	1125	140	12.40%
PC4	Orbiting Sensor Sys.	C	30K	1399	178	12.72%
PC5	Flight Software Plat.	C	164K	17001	503	2.96%

VI. PERFORMANCE METRIC DEFINITIONS

Classification success was extracted via a Standard Binary Confusion Matrix establishing

- True Positive (TP): Defective components correctly tagged as defective.
- False Negative (FN): Defective components misclassified as safe.
- False Positive (FP): Clean components erroneously marked as defective.
- True Negative (TN): Clean components correctly verified as fault-free. [14]

From these elements, we calculate five key evaluation parameters:

1) Accuracy: Proportion of overall correct predictions:

$$(TP+TN)/(TP+TN+FP+FN)$$

2) Sensitivity (Pd): True positive rate or probability of fault detection:

$$TP/(TP+FN)$$

3) Specificity (1-Pf): Accuracy on clean modules:

$$TN/(TN+FP)$$

4) Balance: Euclidean distance between actual model coordinates and coordinates (Pf=0, Pd=1) [14]:

$$\text{Balance} = 1 - [\sqrt{(Pf^2 + (1-Pd)^2)} / \sqrt{2}]$$

5) ROC Area (AUC): Area under the curve assessing classification quality across changing decision thresholds.

VII. Results and Empirical Analysis

The empirical metrics across accuracy, sensitivity, and specificity are mapped below.

TABLE II Global Performance Matrix Across Modules

Repo	Metric	NB	LOG	DT	JRip	PART	J48
CM1	Acc	83.94	87.68	89.13	86.23	73.91	86.23
	Sens	0.400	/0.267	/0.000	/0.200	/0.333	/0.200

	Spec		0.893	0.951	1.000	0.943	0.789	0.943
JM1	Acc	/	81.28	82.02	81.57	81.42	81.13	79.80
	Sens		/0.198	/0.102	/0.070	/0.157	/0.030	/0.131
	Spec		0.813	0.964	0.982	0.959	0.981	0.948
KC1	Acc	/	83.05	86.87	84.84	84.84	83.89	85.56
	Sens		/0.434	/0.238	/0.197	/0.328	/0.320	/0.320
	Spec		0.890	0.984	0.993	0.970	0.959	0.965
KC3	Acc	/	77.50	71.25	75.00	76.25	81.25	80.00
	Sens		/0.412	/0.412	/0.118	/0.118	/0.353	/0.353
	Spec		0.852	0.778	0.921	0.937	0.937	0.921
MC1	Acc	/	94.34	99.27	99.25	99.22	99.19	99.30
	Sens		/0.548	/0.161	/0.194	/0.161	/0.194	/0.161
	Spec		0.946	0.999	0.998	0.998	0.997	0.999
MC2	Acc	/	66.00	66.67	56.86	56.86	70.59	52.94
	Sens		/0.571	/0.545	/0.000	/0.000	/0.500	/0.045
	Spec		0.714	0.758	1.000	1.000	0.849	0.879
MW1	Acc	/	79.25	77.36	85.85	86.79	88.68	85.85
	Sens		/0.429	/0.286	/0.429	/0.143	/0.286	/0.214
	Spec		0.846	0.846	0.923	0.978	0.978	0.956
PC1	Acc	/	88.82	92.11	92.43	89.14	89.80	87.83
	Sens		/0.280	/0.240	/0.160	/0.160	/0.360	/0.240
	Spec		0.935	0.978	0.970	0.949	0.944	0.931
PC2	Acc	/	94.29	99.05	99.37	99.21	99.37	98.90
	Sens		/0.333	/0.000	/0.000	/0.000	/0.000	/0.000
	Spec		0.950	0.998	1.000	1.000	1.000	0.998
PC3	Acc	/	34.38	84.67	80.22	82.89	82.67	82.22
	Sens		/0.986	/0.178	/0.000	/0.233	/0.137	/0.288
	Spec		0.250	0.944	1.000	0.915	0.931	0.899
PC4	Acc	/	87.14	91.79	90.18	90.36	88.21	88.21
	Sens		/0.431	/0.538	/0.231	/0.508	/0.677	/0.692
	Spec		0.935	0.972	0.931	0.963	0.912	0.910
PC5	Acc	/	96.56	96.93	97.01	97.28	96.93	97.13
	Sens		/0.427	/0.308	/0.332	/0.521	/0.474	/0.498
	Spec		0.970	0.982	0.980	0.978	0.973	0.977

VIII. DISCUSSION & ANALYTICAL BREAKDOWN

A. The Sensitivity-Specificity Trade-off

The empirical results reveal a persistent trade-off within static software metrics: algorithms that achieve high global accuracy often struggle with extremely low sensitivity [9]. This behavior points to an underlying limitation in standard predictive models. For instance, in the CM1, MC2, PC2, and PC3 repositories, the Decision Table (DT) miner reports a specificity of exactly 1.000 (100%) but a sensitivity of 0.000 [14].

This occurs because the classifier maps all incoming samples directly to the majority class label. While this strategy yields a high global accuracy score on imbalanced repositories like PC2 (99.37%), it fails to locate actual software bugs, making it impractical for active code verification teams.

B. The Strengths of Bayesian Frameworks

Conversely, Naive Bayes consistently delivers the highest sensitivity scores across 9 out of the 12 evaluated systems [5]. For example, on the PC3 dataset, Naive Bayes captures 98.6% of all defective modules. However, this high detection capability comes with a cost: its specificity drops to 0.250, causing a high rate of false alarms. This behavior stems from the assumption of class conditional independence, which can distort probability estimates when complex, interacting metrics are processed together.

C. Identifying Contextual Top-Performers

No single machine learning model outperforms the rest across all structural configurations [1]. Performance is closely tied to the specific features of the codebase:

1) Object-Oriented Java Structures (KC3): Tree induction tools perform well here, with J48Graft achieving a top accuracy of 82.50% [12].

2) Dense, Highly Imbalanced Datasets (PC4): Decision tree learners excel in this environment [13]. J48 provides a solid balance with a sensitivity of 0.692 and a specificity of 0.910 [12].

Traditional C-Based Structural Modules (MW1): The PART rule engine handles these files most effectively, reaching an accuracy of 88.68% [13].

IX. CONCLUSION & FUTURE WORK

This research evaluated eight machine learning classifiers across twelve NASA MDP repositories using a structured hold-out validation framework [9]. The experimental results show that relying solely on overall accuracy can lead to deceptive models, as high accuracy often masks very low defect detection rates on imbalanced datasets [14]. While Naive Bayes provides excellent initial bug screening due to its high sensitivity, tree and rule-based induction models like J48 and PART offer a more balanced performance for complex systems [12].

Future extensions of this framework should focus on integrating synthetic oversampling techniques (such as SMOTE) to mitigate the impacts of severe class imbalances, evaluating ensemble voting architectures to combine the screening advantages of Bayesian models with the precision of decision trees, and expanding the analysis to include deep learning models to extract insights directly from source code text.

REFERENCES

1. H. Erdogmus, "Data mining static code attributes to learn defect predictors," IEEE Software, vol. 24, no. 1, pp. 12–14, 2007.
2. J. Gondra, "Applying machine learning to software fault-proneness prediction," Journal of Systems and Software, vol. 81, no. 2, pp. 186–195, 2008.
3. B. Yang, K. J. Schon, and X. Liu, "Software defect prediction based on classification rule mining," Information and Software Technology, vol. 53, no. 4, pp. 331–343, 2011.



4. S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahavas, "Regression via classification models for software defect estimation," *Methodologies and Intelligent Systems for Technology*, vol. 22, no. 3, pp. 201–211, 2006.
5. T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.
6. A. D. Oral and A. B. Bener, "Defect prediction for embedded software systems," *Computer and Information Sciences*, vol. 22, pp. 1–6, 2007.
7. Y. Chen, X. Shen, P. Du, and B. Ge, "Research on software defect prediction based on data mining architectures," *Proceedings of the International Conference on Computer Engineering*, vol. 1, pp. 563–567, 2010.
8. M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect data sets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.
9. R. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.
10. Q. Song, M. Jia, M. Ying, and H. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, 2011.
11. G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using Bayesian methods," *IEEE Transactions on Software Engineering*, Vol 33 no. 10 – 2008
12. M. Hall, E. Frank, G. Holmes, "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009
13. D. M. W. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011