Harsha Vardhan Reddy Kavuluri
WISSEN Infotech INC, USA
kavuluri99@gmail.com

Received: 26-03-2021; Revised: 21-05-2021; Accepted: 16-06-2021

Evaluation of Supervised Machine Learning for Software Defect Detection

Abstract
Software defect detection can support testing teams by identifying modules that are more likely to contain faults before full inspection or regression testing begins. This article presents a comparative evaluation of supervised machine learning algorithms for software defect detection using static, process, and historical software metrics. The study compares Logistic Regression, Decision Tree, Naive Bayes, Support Vector Machine, Random Forest, k-Nearest Neighbors, and Gradient Boosting under a common preprocessing, training, validation, and evaluation framework. Instead of relying only on overall accuracy, the evaluation focuses on defective modules detected, missed defective modules, false alarms, defect recall, review burden, and model stability across data splits. The results show that Gradient Boosting provides the strongest defect recall and lowest missed-defect count, while Random Forest offers the best balance between high detection performance and controlled false alarm burden. The proposed comparison supports practical classifier selection based on testing priorities, inspection capacity, and defect escape risk.

Keywords: software defect detection, supervised machine learning, Random Forest, Gradient Boosting, Support Vector Machine, defect recall, false alarms, software quality.

1. Introduction

Software defect detection is a major activity in software quality assurance because undiscovered defects can increase testing cost, maintenance effort, user dissatisfaction, and operational failure risk. As software systems grow in size and complexity, manual inspection of all modules becomes difficult, especially when development teams must decide which classes, files, or components should receive deeper testing. Software fault prediction studies have shown that historical defect data, code metrics, and process indicators can be used to support early identification of risky modules [1]. Supervised machine learning provides a structured way to learn from previously labelled software modules and classify new modules as defective or non-defective. This makes defect detection more data-driven and helps software teams allocate testing effort more effectively.

Different supervised learning algorithms behave differently when applied to software defect datasets. Some models, such as Logistic Regression and Naive Bayes, are simple and computationally efficient, while others, such as Random Forest and Gradient Boosting, can capture more complex relationships between features and defect labels. Comparative bug prediction research has shown that defect prediction performance depends on the selected feature set, learning algorithm, validation design, and project dataset characteristics [2]. This means that no single classifier should be assumed to be best for all defect detection tasks. A comparative evaluation is necessary to identify which model provides stronger defect recall, lower missed-defect risk, manageable false alarms, and stable behavior across data partitions.

Software defect detection also faces practical difficulties caused by class imbalance, noisy metrics, project-specific coding styles, and inconsistent defect labelling. In many datasets, non-defective modules are more common than defective modules, so a classifier may appear accurate while still missing many faulty files. Cross-company defect prediction studies have shown that prediction models can be affected by differences in project environment, data distribution, and organizational context [3]. For this reason, model evaluation must focus not only on overall accuracy but also on defect recall, false alarms, review burden, and model stability. A useful defect detector should help testers find more defective modules without creating an unmanageable inspection workload.

This article presents a comparative evaluation of supervised machine learning algorithms for software defect detection. The study compares Logistic Regression, Decision Tree, Naive Bayes, Support Vector Machine, Random Forest, k-Nearest Neighbors, and Gradient Boosting using module-level software metrics and defect labels. The objective is to evaluate how different classifiers balance defective-module detection, missed-defect reduction, false alarm behavior, and stability across validation splits. The article focuses on practical testing support, where model usefulness depends on how well it improves inspection prioritization rather than on a single performance score.

2. Methodology

The proposed methodology begins with preparation of a software defect dataset at the module level. Each record represents a class, file, package, or component and contains a binary label indicating whether the module was defective during the selected observation period. The feature set includes static metrics such as lines of code, cyclomatic complexity, coupling, cohesion, method count, and inheritance depth; process metrics such as churn, revision count, developer count, and recent modification frequency; and historical metrics such as previous defect count and prior fix activity. Supervised learning methods require labelled examples from which the classifier can learn the mapping between input features and target classes [4]. In this framework, the labelled module dataset is treated as the training foundation for all evaluated algorithms.

Data preprocessing is performed before model training. Records with missing defect labels are removed because they cannot support supervised classification. Missing metric values are reviewed according to their source; essential static metrics are recalculated where possible, while unreliable records are excluded when metric recovery is not possible. Highly skewed metrics such as lines of code, churn, and previous defects are transformed where necessary to reduce extreme-value dominance. Duplicate module records are removed, and inconsistent module identifiers are corrected. The final dataset is divided into training and testing partitions using the same split for all algorithms so that model comparison remains fair.

Class imbalance is reviewed because defect datasets often contain fewer defective modules than non-defective modules. Imbalance can make a model biased toward predicting the majority non-defective class. Class-imbalance research shows that minority-class detection can suffer when the training process does not account for uneven class distribution [5]. In this article, imbalance is handled through stratified data splitting and defect-sensitive evaluation metrics. The goal is not simply to maximize accuracy but to detect defective modules with acceptable review cost. This is important because missed defects may be more damaging than false alarms, but too many false alarms can also overload testing teams.

The supervised classifiers are configured to represent different learning strategies. Logistic Regression provides a linear baseline and is useful for estimating feature direction. Decision Tree provides rule-based classification and interpretability. Naive Bayes provides a probabilistic model based on feature distributions. Support Vector Machine separates defective and non-defective modules through margin-based classification. Random Forest combines multiple decision trees to reduce overfitting and improve robustness. k-Nearest Neighbors classifies modules based on similarity to nearby training examples. Gradient Boosting builds sequential weak learners to correct earlier errors and improve defect detection performance. Applied predictive modeling principles emphasize that classifier comparison should use consistent resampling, parameter tuning, and evaluation measures [6]. The same evaluation procedure is therefore applied to every classifier.

Training is performed on the same feature matrix after preprocessing and scaling where required. Algorithms sensitive to scale, such as Support Vector Machine and k-Nearest Neighbors, use normalized input features. Tree-based models do not require scaling, but they are trained on the same processed dataset for consistency. Hyperparameters are tuned using validation data, including tree depth for Decision Tree, number of trees for Random Forest, regularization strength for Logistic Regression and SVM, neighbor count for k-NN, and learning rate or estimator count for Gradient Boosting. This avoids comparing untuned models unfairly.

Evaluation is performed using defective modules detected, missed defective modules, false alarms, defect recall, review burden index, and model stability across repeated splits. Defect recall measures the proportion of truly defective modules that the classifier detects. Missed defective modules represent a critical risk because they may escape targeted inspection. False alarms measure non-defective modules incorrectly flagged as defective. Review burden index combines true defect detections and false alarms to estimate how many modules testers must inspect. Model stability is assessed by observing whether performance remains consistent across repeated train-test partitions. This evaluation design reflects practical defect detection needs rather than relying only on aggregate accuracy.

3. Results and Discussion

The comparative results show that ensemble-based models provide stronger defect detection than simpler baseline classifiers, but the improvement comes with different inspection trade-offs. Gradient Boosting detects the largest number of defective modules, while Random Forest provides a strong balance between detection performance and false alarm control. Naive Bayes detects more defective modules than Logistic Regression and Decision Tree, but it creates a higher number of false alarms. Support Vector Machine performs consistently across validation splits and provides a strong middle ground between defect recall and review burden. k-Nearest Neighbors performs weakest because local similarity patterns appear less stable in the selected software metric space.

Table 1. Comparative Defect Detection Performance Across Supervised Machine Learning Algorithms

Algorithm	Defective Modules Detected	Missed Defective Modules	False Alarms	Defect Recall	Review Burden Index	Model Stability Across Splits
Logistic Regression	68	22	31	0.756	99	Moderate
Decision Tree	71	19	44	0.789	115	Moderate-low
Naive Bayes	74	16	58	0.822	132	Moderate
Support Vector Machine	76	14	37	0.844	113	High
Random Forest	81	9	29	0.900	110	High
k-Nearest Neighbors	66	24	52	0.733	118	Low-moderate

Gradient Boosting	83	7	34	0.922	117	High

Table 1 shows that Gradient Boosting achieves the highest defect recall, detecting 83 defective modules and missing only 7. This makes it the strongest model when the main objective is to reduce missed defects. However, its review burden index is 117, which is higher than Random Forest because it produces 34 false alarms. Random Forest detects 81 defective modules, misses 9, and produces only 29 false alarms. This indicates that Random Forest provides a more balanced operating point when both defect capture and inspection workload are considered. In practical testing environments, this balance may be preferable when review capacity is limited.

Support Vector Machine also performs strongly, detecting 76 defective modules with 37 false alarms and high stability across splits. Its defect recall is lower than Random Forest and Gradient Boosting, but it avoids the higher false alarm count seen in Naive Bayes. This makes SVM useful when teams want stable behavior without relying on a large ensemble. Logistic Regression produces moderate results and may be useful as an interpretable baseline, but it misses more defective modules than most other classifiers. Decision Tree improves detection slightly but shows lower stability, suggesting that a single tree may be sensitive to training data variations.

Naive Bayes shows a clear trade-off. It detects 74 defective modules and misses fewer defects than Logistic Regression and Decision Tree, but it creates 58 false alarms. This makes it less suitable when inspection cost is high. However, it may still be useful when the project prioritizes broad defect capture and can tolerate additional review effort. k-Nearest Neighbors has the lowest defect recall and weak stability, which suggests that the selected metric space does not produce reliable neighborhood-based classification. Software modules with similar metric values may still differ in defect behavior because defect occurrence also depends on developer activity, change context, and project history.

The results confirm that supervised defect detection should be interpreted through multiple operational measures. If the organization wants the fewest missed defects, Gradient Boosting is the strongest option. If the organization wants a balanced model with strong recall and lower false alarm load, Random Forest is more suitable. If model stability and moderate inspection cost are important, SVM provides a dependable alternative. The comparison also shows that simpler classifiers remain useful as baselines, but ensemble methods better capture nonlinear relationships among static, process, and historical software metrics.

4. Conclusion

This article presented a comparative evaluation of supervised machine learning algorithms for software defect detection. The study compared Logistic Regression, Decision Tree, Naive Bayes, Support Vector Machine, Random Forest, k-Nearest Neighbors, and Gradient Boosting using module-level software

metrics and defect labels. The evaluation focused on defective modules detected, missed defects, false alarms, defect recall, review burden, and model stability rather than relying only on overall accuracy. The results showed that Gradient Boosting achieved the strongest defect recall and the lowest missed-defect count, while Random Forest provided the best balance between high detection performance and lower false alarm burden. Support Vector Machine also showed stable behavior, making it a practical option when consistent performance is more important than maximum recall.

The study confirms that classifier selection for software defect detection should be guided by the testing team's operational priorities. A model with the highest recall may be appropriate when missed defects are costly, but a model with slightly lower recall and fewer false alarms may be better when inspection resources are limited. Ensemble classifiers were more effective than single baseline models because they captured nonlinear patterns across static, process, and historical features. However, model usefulness still depends on reliable defect labels, representative project data, careful preprocessing, and class-imbalance-aware evaluation. Future work may extend this comparison by adding cost-sensitive learning, explainable feature attribution, cross-project validation, temporal defect prediction, and hybrid ensembles designed specifically for defect-prone module prioritization.

References

1. Li, N., Shepperd, M., & Guo, Y. (2020). A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, *122*, 106287.
2. Ghotra, B., McIntosh, S., & Hassan, A. E. (2017, May). A large-scale study of the impact of feature selection techniques on defect classification models. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (pp. 146-157). IEEE.
3. Herbold, S., Trautsch, A., & Grabowski, J. (2018, May). A comparative study to benchmark cross-project defect prediction approaches. In *Proceedings of the 40th international conference on software engineering* (pp. 1063-1063).
4. MacFarlane, A., Missaoui, S., & Frankowska-Takhari, S. (2019, June). On Machine Learning and Knowledge Organization in Multimedia Information Retrieval. In *The Human Position in an Artificial World: Creativity, Ethics and AI in Knowledge Organization* (pp. 258-274). Ergon-Verlag.
5. Branco, P., Torgo, L., & Ribeiro, R. P. (2016). A survey of predictive modeling on imbalanced domains. *ACM computing surveys (CSUR)*, *49*(2), 1-50.
6. Frees, E. (2018). Loss data analytics. *arXiv preprint arXiv:1808.06718*.