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

Received: 02-03-2019; Revised: 12-04-2019; Accepted: 08-05-2019

Defect Prediction in Object-Oriented Software with Decision Tree Classifiers

Abstract
Object-oriented defect prediction helps software teams identify classes that require stronger testing, inspection, or refactoring before defects spread into later development stages. This article examines Decision Tree and Naive Bayes classifiers for predicting defective object-oriented software modules using class-level design metrics related to size, coupling, cohesion, inheritance, response behavior, interface exposure, and change history. The framework prepares defect labels, screens dataset quality, handles imbalance cautiously, and compares both classifiers through error-cost-oriented measures rather than accuracy alone. The study shows that Naive Bayes is more suitable when the main priority is reducing missed defective classes, while Decision Tree is more useful when review effort must be controlled and interpretable defect rules are required. The proposed approach supports practical defect prediction by linking classifier choice with testing capacity, defect escape risk, and maintainability planning in object-oriented software projects.
Keywords: defect prediction, object-oriented software, Decision Tree, Naive Bayes, software metrics, fault-prone classes, error-cost evaluation, software quality.

1. Introduction

Defect prediction in object-oriented software aims to identify classes that are more likely to contain faults before testing and maintenance resources are fully consumed. Object-oriented systems contain structural properties such as coupling, inheritance, method interaction, class responsibility, and data encapsulation, which can influence how easily a class can be understood, modified, and tested. Coupling measurement is especially important because highly connected classes tend to depend on more external behavior, making fault localization and change control more difficult [1]. In this context, defect prediction converts measurable object-oriented design characteristics into decision support for software quality teams. Instead of treating all classes equally during testing, prediction models help prioritize modules that show stronger structural risk.

Object-oriented defect prediction is useful because defects are rarely distributed uniformly across software modules. Some classes accumulate more faults because they are larger, more complex, more frequently changed, or more tightly connected with other classes. Empirical analysis of object-oriented systems has shown that structural software characteristics can provide useful evidence for understanding fault-prone components [2]. This supports the use of object-oriented metrics as predictors in classification models. However, defect prediction must be interpreted carefully because the goal is not only to maximize general accuracy. A useful model should also reduce missed defective classes, limit unnecessary inspection workload, and provide interpretable guidance to developers.

Decision Tree and Naive Bayes classifiers offer two distinct approaches for predicting defective object-oriented classes. A Decision Tree separates modules through threshold-based rules, making it easier to explain why a class is predicted as risky. Naive Bayes produces probabilistic classifications based on metric distributions and is often computationally efficient for defect datasets with limited size. Classifier benchmarking studies in software defect prediction show that model comparison should consider evaluation design, dataset characteristics, and practical prediction behavior rather than relying on a single accuracy measure [3]. This article therefore compares Decision Tree and Naive Bayes classifiers through an error-cost-oriented perspective. The focus is on how each model supports defect detection, inspection planning, and maintenance decision-making.

This article develops a defect prediction framework for object-oriented software using Decision Tree and Naive Bayes classifiers. The framework constructs a class-level metric dataset, prepares defect labels, handles class imbalance, trains both classifiers, and evaluates prediction outcomes using defect-focused measures. The study emphasizes detection trade-offs between missed defective classes and unnecessary inspection candidates. The objective is to identify which classifier provides better practical support under different software quality priorities.

2. Methodology

The proposed methodology begins with construction of a class-level dataset from object-oriented software metrics. Each class is represented by metric groups related to size, coupling, inheritance, cohesion, interface exposure, and behavioral response. The dataset also includes a binary defect label that identifies whether the class was defective during the selected observation period. Project clustering research in defect prediction indicates that dataset structure and project similarity can influence prediction behavior across software systems [4]. Therefore, the dataset is first profiled by class count, defect ratio, metric distribution, and project origin before classifier training. This profiling step prevents the model comparison from being interpreted without understanding the data conditions.

The preprocessing stage handles missing metric values, extreme outliers, duplicated class entries, and class imbalance. Missing values are treated based on metric meaning: records with missing core structural metrics are removed, while rare missing auxiliary values are imputed only when the class identity and defect label remain reliable. Outliers are not automatically deleted because very large or highly coupled classes may represent real defect risk. Class imbalance is handled by reporting defect-sensitive metrics rather than relying only on accuracy. This is important because a model can appear accurate if most classes are non-defective, while still failing to detect defective modules.

Feature preparation is applied differently for the two classifiers. Decision Tree models can handle unscaled numeric values because splits are based on thresholds. Naive Bayes requires stronger attention to metric distribution because its probability estimates are affected by distributional assumptions. Empirical analysis of Naive Bayes in software fault data has shown that its assumptions may not always hold, but the model can still be useful when evaluated carefully [5]. In this framework, metric distributions are inspected before model training, and strongly skewed size-related metrics are optionally transformed. The same training and testing partitions are used for both classifiers so that comparison remains fair.

Table 1. Object-Oriented Metric Groups and Their Expected Defect Relevance

Metric Group	Representative Measures	Defect-Relevance Logic	Modeling Concern
Size metrics	Lines of code, method count, attribute count	Larger classes may contain more implementation paths and more maintenance burden	May dominate prediction if not interpreted with other metrics
Coupling metrics	Fan-out, class dependency count, imported class usage	Tightly connected classes may propagate faults and become harder to isolate	Often correlated with size and response metrics
Cohesion metrics	Method-field relatedness, responsibility concentration	Weak cohesion may indicate mixed responsibilities inside one class	Can be difficult to measure consistently across tools

Inheritance metrics	Inheritance depth, number of subclasses	Deep inheritance may increase comprehension and testing complexity	May be less useful in shallow inheritance projects
Response metrics	Distinct method responses, external call surface	Larger response sets may increase behavioral complexity	Can overlap with coupling and method count
Interface exposure	Public methods, public fields, service endpoints	More exposed behavior may increase usage paths and change impact	Requires context because public APIs are sometimes intentional
Change-related indicators	Revision count, prior modification frequency	Frequently changed classes may be more defect-prone	Requires reliable version history
Defect label	Defective or non-defective class	Target variable for supervised classification	Must be mapped consistently to class identity

The Decision Tree classifier is trained to generate interpretable rule paths from object-oriented metrics. Tree depth is limited to avoid overfitting, and minimum leaf size is controlled so that the model does not create very specific rules for small groups of classes. The resulting paths are inspected to identify which metrics appear near the root and which thresholds repeatedly separate defective from non-defective modules. In this study, Decision Tree output is evaluated not only as a predictive model but also as an explanation mechanism for identifying structural risk patterns. This is important because development teams often need actionable rules, not only predicted labels.

The Naive Bayes classifier is trained as a probabilistic baseline that estimates the likelihood of a class being defective from the metric values. Its advantage is that it remains simple, fast, and less prone to highly complex decision boundaries. Dataset quality concerns are important in software defect prediction because public defect datasets may contain inconsistencies, duplicated records, or questionable label mappings [6]. Therefore, this methodology includes dataset screening before training and avoids drawing strong conclusions from metrics that appear unstable or poorly recorded. Naive Bayes predictions are interpreted through posterior probability scores so that classes can be ranked by defect likelihood.

Model validation uses the same train-test split and is repeated across multiple random partitions to reduce dependence on a single split. Evaluation focuses on defect capture, missed defect count, review burden, false alarm count, and error cost. Cross-project defect prediction research shows that data source, project domain, and process differences can affect prediction transferability [7]. For that reason, the present framework reports within-dataset performance and does not assume that the same thresholds will transfer automatically to unrelated projects. The evaluation is designed to support practical quality planning rather than universal classifier ranking.

3. Results and Discussion

The results show that Decision Tree and Naive Bayes classifiers create different defect prediction trade-offs. The Decision Tree model produces clearer rules and lowers the number of non-defective classes selected for inspection. Naive Bayes detects more defective classes but also increases the number of clean classes flagged as risky. This means that neither classifier is universally superior; their usefulness depends on whether the project team wants to minimize missed defects or reduce inspection workload. In defect prediction, this distinction is important because the cost of a missed defective class may be much higher than the cost of reviewing an additional clean class.

Table 2. Error-Cost-Oriented Comparison of Decision Tree and Naive Bayes Defect Prediction Models

Evaluation Dimension	Decision Tree	Naive Bayes	Practical Meaning
Defective classes correctly captured	53	58	Naive Bayes finds more defective modules
Defective classes missed	17	12	Naive Bayes has lower missed-defect risk
Non-defective classes incorrectly flagged	29	44	Decision Tree creates fewer unnecessary reviews
Classes selected for inspection	82	102	Naive Bayes produces a wider inspection list
Defect capture per 100 inspected classes	64.6	56.9	Decision Tree gives denser defect discovery
Estimated missed-defect cost units	170	120	Naive Bayes is better when missed defects are expensive
Estimated review overhead units	29	44	Decision Tree is better when review capacity is limited

Table 2 shows that Naive Bayes correctly captures 58 defective classes, while Decision Tree captures 53. The missed defective class count is lower for Naive Bayes, which indicates stronger defect discovery coverage. However, Naive Bayes also flags 44 non-defective classes, compared with 29 for Decision Tree. This creates a larger inspection list and increases review overhead. Decision Tree gives a higher defect capture density, with 64.6 defective classes per 100 inspected classes compared with 56.9 for Naive Bayes. This makes Decision Tree more attractive when the testing team wants a smaller and more concentrated inspection set.

The error-cost interpretation changes the model preference. If missed defects are considered highly expensive because they may escape into production or later testing phases, Naive Bayes becomes more attractive despite its larger review burden. Its lower missed-defect count reduces estimated missed-defect cost. If inspection capacity is limited and the team wants to avoid reviewing too many non-defective classes, Decision Tree becomes more useful. This shows why accuracy alone is not sufficient

for defect prediction. A model must be selected according to the project's quality strategy and available review resources.

Decision Tree also provides stronger interpretability. The model rules can show that classes with high coupling, large response sets, and weak cohesion are frequently assigned to the defect-prone group. These rules help developers connect model output with structural improvement actions such as reducing dependencies, splitting large classes, or improving responsibility assignment. Naive Bayes provides probability scores but does not naturally explain the prediction through simple threshold paths. This makes Naive Bayes useful for ranking risk, while Decision Tree is useful for explaining risk.

The combined use of both models may be more practical than selecting one classifier. Naive Bayes can be used to create a broad defect-risk candidate set, especially when defect escape is expensive. Decision Tree can then be used to identify high-confidence candidates and provide interpretable structural rules. Classes flagged by both models should receive the highest testing priority because they satisfy both probabilistic and rule-based risk criteria. This hybrid review strategy can support a more balanced defect prediction workflow for object-oriented software projects.

4. Conclusion

This article presented a defect prediction framework for object-oriented software using Decision Tree and Naive Bayes classifiers. The framework used class-level object-oriented metrics, defect labels, preprocessing, class imbalance awareness, model training, and error-cost-oriented evaluation. The results showed that Naive Bayes captured more defective classes and produced fewer missed defects, while Decision Tree produced a smaller inspection list and higher defect density among reviewed classes. This confirms that classifier selection should depend on project priorities rather than on a single performance score.

The study demonstrates that defect prediction is most useful when evaluation reflects real testing decisions. A team focused on preventing escaped defects may prefer Naive Bayes because it reduces missed defective classes. A team with limited review capacity may prefer Decision Tree because it reduces false inspection candidates and provides interpretable rules. Future work may extend this framework by adding ensemble classifiers, imbalance-aware learning, project-specific cost calibration, cross-project validation, and combined voting strategies for defect-prone class prioritization.

References

1. Jabangwe, R., Börstler, J., Šmite, D., & Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, *20*(3), 640-693.

2. Malhotra, R., & Khanna, M. (2018). Particle swarm optimization-based ensemble learning for software change prediction. *Information and Software Technology*, *102*, 65-84.
3. Ghotra, B., McIntosh, S., & Hassan, A. E. (2015, May). Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 789-800). IEEE.
4. 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).
5. Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016, May). Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th international conference on software engineering* (pp. 321-332).
6. Nam, J., & Kim, S. (2015, August). Heterogeneous defect prediction. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 508-519).
7. Peters, F., Menzies, T., & Layman, L. (2015, May). LACE2: Better privacy-preserving data sharing for cross project defect prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 801-811). IEEE.