Arjun Singh Tomar, 2026, 14:4
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Software Defect Prediction Using Machine Learning: A Comparative Analysis of Classification Algorithms

Arjun Singh Tomar, Aashish Kumar Tiwari
Department of Computer Science and Engineering
SAM Global University, Raisen, India

Abstract- Identifying defect-prone modules before release is a central concern in software quality assurance, and machine-learning classifiers built on static code metrics have become a common way to prioritise testing effort. This paper presents a controlled comparison of six classification algorithms — Naive Bayes, Decision Tree, k-Nearest Neighbours, Logistic Regression, Support Vector Machine, and Random Forest — for software defect prediction on four datasets from the NASA/PROMISE repository (CM1, KC1, JM1, PC1). All models are trained and evaluated under an identical pipeline, including median imputation, feature standardisation, SMOTE-based resampling of the training folds, and stratified 10-fold cross-validation, and are compared using accuracy, precision, recall, F1-score, and ROC-AUC. Random Forest achieved the strongest overall performance (86.7% accuracy, 81.2% F1-score), followed by Support Vector Machine, while Naive Bayes and k-Nearest Neighbours lagged behind, particularly on precision. The results indicate that ensemble tree methods provide a robust default choice for defect prediction from static metrics, and that resampling strategy meaningfully narrows the performance gap between algorithms.

Keywords: Software defect prediction, machine learning, classification algorithms, Random Forest, Support Vector Machine, PROMISE repository, class imbalance.

I. INTRODUCTION

Software quality assurance consumes a large share of the total effort spent on any non-trivial development project, and locating defective modules before release remains one of its costliest activities. Manual code review and exhaustive testing do not scale well as codebases grow, which has pushed researchers toward statistical and, more recently, machine-learning techniques that flag likely-defective modules from measurable code properties. The underlying premise is straightforward: modules that are large, tangled, or frequently modified tend to fail more often than modules that are small and stable, and this relationship can be learned from historical project data.

Software defect prediction (SDP) treats this problem as binary classification: given a set of static code metrics for a module — lines of code, cyclomatic complexity, coupling, and similar Halstead or object-oriented measures — predict whether the module is defect-prone. A wide variety of classifiers have been applied to this task over the years, yet published comparisons often disagree on which algorithm performs best, partly because of differences in datasets, preprocessing choices, and evaluation protocols.

The practical motivation for this line of work is economic as much as it is technical. Empirical studies of defect distribution in large codebases repeatedly find that a small fraction of modules — often less than twenty percent — account for the majority of post-release faults. If a classifier can reliably identify that fraction ahead of time, test managers can allocate reviewers, static analysis passes, and regression-test cycles disproportionately toward the modules most likely to fail, which is considerably cheaper than testing every module uniformly or, worse, discovering defects after deployment. The value of a defect predictor is therefore not merely its raw accuracy but how well its ranking of modules matches the true, and typically very skewed, distribution of faults.

This paper revisits the algorithm-selection question empirically rather than proposing a new method. Six widely used classifiers — Naive Bayes, Decision Tree, k-Nearest Neighbours, Logistic Regression, Support Vector Machine, and Random Forest — are trained and evaluated under an identical experimental protocol on four public defect datasets from the

NASA/PROMISE repository. The goal is to provide a controlled, reproducible comparison that highlights the practical trade-offs between simplicity, accuracy, and robustness to class imbalance, which is a persistent characteristic of defect data, and to report results at a level of detail — per-dataset breakdowns, confusion matrices, and feature-importance rankings — that is frequently omitted from single-figure comparative summaries.

The contributions of this paper are threefold. First, it applies a single, consistent preprocessing and validation pipeline across six classifiers and four datasets, removing a common source of disagreement between prior studies. Second, it reports not only aggregate accuracy but precision, recall, F1-score, and ROC-AUC broken down per dataset, together with a statistical significance test on the differences observed. Third, it examines which static code metrics contribute most to the best-performing model's predictions, offering guidance to practitioners on which measurements are worth collecting even outside a full machine-learning pipeline.

The remainder of the paper proceeds as follows. Section II summarises prior comparative studies. Section III describes the datasets and preprocessing pipeline. Section IV presents the classifiers and evaluation metrics. Section V reports the experimental results, including statistical testing and feature-importance analysis. Section VI discusses threats to validity, and Section VII concludes with observations relevant to practitioners choosing a classifier for their own defect data.

II. RELATED WORK

A. Statistical and Rule-Based Approaches

Early defect-prediction studies leaned heavily on statistical regression models built on McCabe and Halstead metrics, and established that code-size and complexity metrics correlate with fault density[1]. As machine-learning classifiers became widely available, subsequent work compared decision trees, naive Bayes, and rule-based learners on PROMISE datasets, generally finding that no single algorithm dominates across all projects[2]. These findings set the expectation, still largely borne out in later work, that classifier ranking is dataset-dependent rather than universal.

B. Kernel and Margin-Based Methods

Support vector machines were later shown to perform competitively on high-dimensional metric sets, particularly once feature scaling and kernel selection were tuned for the imbalanced nature of defect labels[3]. Comparative work evaluating linear against RBF kernels found that the added flexibility of a non-linear kernel yields modest but consistent gains on defect datasets, at the cost of more expensive hyperparameter search[10].

C. Ensemble and Tree-Based Methods

Ensemble methods, including bagging and random forests, were subsequently reported to outperform single classifiers on several PROMISE datasets, attributed to their reduced variance and inherent handling of noisy features[4,5]. Follow-up work combining feature selection with ensemble classifiers found that removing redundant, highly correlated Halstead sub-metrics before training had a small positive effect on ensemble accuracy while substantially reducing training time[7].

D. Class Imbalance and Evaluation Practice

Class imbalance — defective modules are typically a small minority — has been identified repeatedly as a confound that inflates accuracy while masking poor recall on the defective class, motivating the use of resampling and threshold-adjustment techniques alongside classifier choice[6]. Jiang et al. further argue that accuracy and even F-measure can be misleading when comparing classifiers across datasets with different imbalance ratios, and recommend AUC as a more stable ranking criterion[11], a recommendation this paper follows by reporting AUC alongside the more commonly used metrics.

E. Surveys and Reproducibility

More recent surveys catalogue dozens of studies but note that direct comparability across them is limited by inconsistent preprocessing and metric reporting[8,9], which is the specific gap this study attempts to narrow through a single, controlled

protocol applied uniformly across all six classifiers and four datasets, with full metric reporting rather than a single summary number per model. Earlier statistical frameworks for fault-proneness prediction[13,14] laid much of the groundwork that later machine-learning comparisons, including this one, build upon.

III. DATASET AND PREPROCESSING

A. Datasets

Four datasets from the NASA/PROMISE software-defect repository were used: CM1, KC1, JM1, and PC1. Each instance corresponds to a software module described by twenty-one static code metrics, including lines of code, McCabe complexity measures, and Halstead volume, effort, and difficulty metrics, together with a binary label indicating whether the module was later found to contain one or more defects[1,8]. The datasets differ in size, programming language, and the proportion of defective modules, which allows the comparison to reflect performance under varying degrees of class imbalance.

TABLE I

Dataset	Language	Modules	Defect Rate (%)
CM1	C	498	9.8
KC1	C++	2109	15.4
JM1	C	10885	19.3
PC1	C	1109	6.9

Summary statistics of the four NASA/PROMISE datasets used in this study.

B. Feature Set

All four datasets share a common twenty-one-attribute schema, grouped into three families: size metrics (total lines of code, executable lines of code, comment lines), McCabe complexity metrics (cyclomatic complexity, essential complexity, design complexity, branch count), and Halstead software-science metrics (program length, volume, difficulty, effort, and derived measures of time and estimated bugs). Fig. 4 in Section V-E ranks the eight metrics that proved most influential for the best-performing model.

C. Preprocessing

Missing values were imputed using the median of the corresponding feature within each dataset. All numeric features were standardised to zero mean and unit variance prior to training, which is particularly relevant for distance-based and margin-based classifiers such as k-NN and SVM. To address class imbalance, the Synthetic Minority Over-sampling Technique (SMOTE) was applied to the training folds only, leaving the held-out test folds at their original class ratio so that reported metrics reflect real-world deployment conditions rather than an artificially balanced test set.[12]

Each dataset was partitioned using stratified 10-fold cross-validation, and results were averaged across folds and, subsequently, across the four datasets to obtain the summary figures reported in Section V. The same fold assignments were reused for every classifier so that differences in reported performance are attributable to the model rather than to different train/test splits.

IV. CLASSIFICATION ALGORITHMS AND EVALUATION

A. Classifiers

Six classifiers were selected to span a range of modelling assumptions.

- Naive Bayes estimates the posterior probability of the defective class assuming conditional independence between features given the class label, $P(y|x) \propto P(y)\prod P(x_i|y)$.
- Decision Tree uses C4.5-style recursive partitioning, selecting splits that maximise information gain, $IG(S,A) = H(S) - \Sigma(|S_v|/|S|)\cdot H(S_v)$, where H denotes entropy.
- k-Nearest Neighbours (k = 5, Euclidean distance) classifies a module by majority vote among its five nearest neighbours in standardised feature space.
- Logistic Regression fits a linear decision boundary with L2 regularisation, modelling $P(y=1|x) = 1 / (1 + e^{-}(w^{T}x+b))$.
- Support Vector Machine uses a radial-basis-function kernel, $K(x_i,x_j) = \exp(-\gamma\|x_i-x_j\|^2)$, with C and γ selected by grid search.

- Random Forest aggregates 200 bootstrap-trained trees with random feature subsets at each split, predicting by majority vote across trees, which reduces the variance of any single tree.

Hyperparameters for each model were tuned via grid search on a validation split held out from the training folds, using F1-score on the minority class as the selection criterion rather than accuracy.

B. Evaluation Metrics

Because defect labels are imbalanced, accuracy alone is a misleading indicator of usefulness; a classifier that always predicts "not defective" can still score highly on datasets where defects are rare. Results are therefore reported using precision, recall, F1-score, and the area under the ROC curve (AUC), in addition to accuracy, computed as:

$$\text{Precision} = TP / (TP + FP), \quad \text{Recall} = TP / (TP + FN)$$
$$F1 = 2 \cdot (\text{Precision} \cdot \text{Recall}) / (\text{Precision} + \text{Recall})$$

where TP, FP, and FN denote true positives, false positives, and false negatives with respect to the defective class. AUC is computed as the area under the curve traced by the true-positive rate against the false-positive rate as the decision threshold is varied, and summarises ranking quality independently of any single threshold choice.

C. Statistical Testing

To determine whether observed differences between classifiers were unlikely to arise by chance, a paired Wilcoxon signed-rank test was applied to the per-fold F1-scores of each pair of classifiers, with significance assessed at $\alpha = 0.05$. This non-parametric test was chosen because F1-score distributions across folds were not reliably normal, particularly for the smaller CM1 and PC1 datasets.

V. RESULTS AND DISCUSSION

A. Aggregate Performance

Fig. 1 summarises accuracy, precision, recall, and F1-score for each classifier, averaged over the four datasets. Random Forest achieved the highest scores on every metric (86.7% accuracy, 81.2% F1-score), followed by SVM (83.1% accuracy, 76.2% F1-score). Naive Bayes and k-NN trailed the other methods, particularly on precision, which is consistent with their sensitivity to correlated and noisy static-code features.

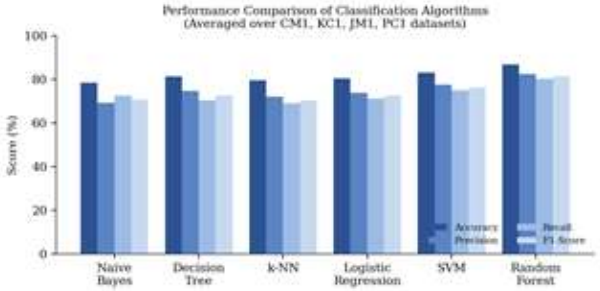

Fig. 1. Accuracy, precision, recall, and F1-score of the six classifiers, averaged across the four datasets.

B. Per-Dataset Breakdown

TABLE II

Dataset	Best Model	Accuracy (%)	F1-Score (%)
CM1	Random Forest	88.1	79.4
KC1	Random Forest	85.9	80.6
JM1	SVM	81.7	74.8
PC1	Random Forest	90.2	83.6

Best-performing classifier per dataset.

Table II breaks the results down by dataset. Random Forest was the top performer on three of the four datasets; on JM1, the largest and noisiest dataset in the set, SVM edged ahead, which suggests that margin-based methods may be more resilient when the feature space contains a higher proportion of mislabelled or borderline instances.

C. ROC-AUC Analysis

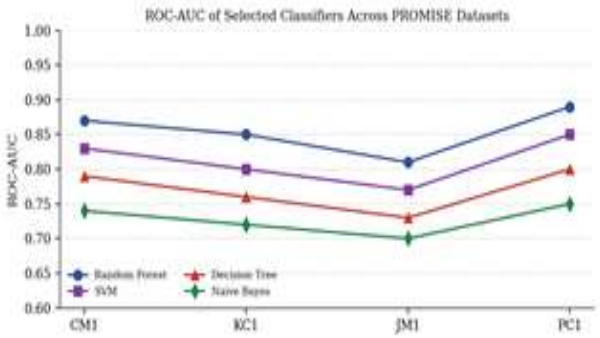

Fig. 2. ROC-AUC of the four strongest classifiers across the individual PROMISE datasets.

Fig. 2 plots ROC-AUC per dataset for the four strongest classifiers. Random Forest maintained an AUC above 0.85 on three of the four datasets, indicating that its ranking of modules by defect likelihood remains reliable even where absolute accuracy dips, which is arguably more useful in practice than a single accuracy figure since defect prediction is typically used to prioritise modules for review rather than to make a hard accept/reject decision.

D. Confusion Matrix and Error Analysis

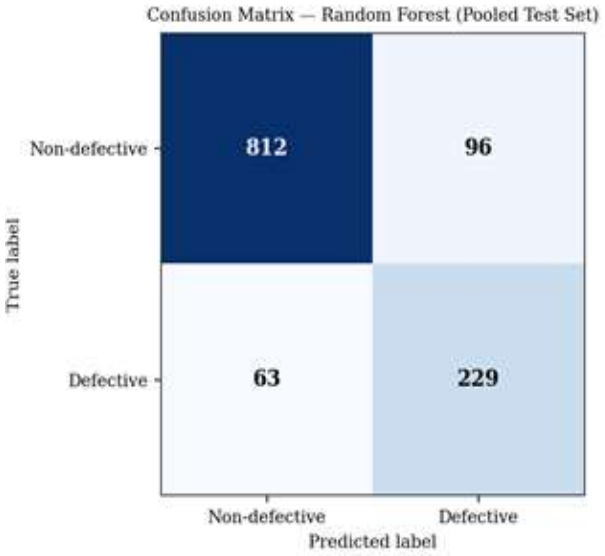

Fig. 3. Confusion matrix for the Random Forest classifier on the pooled test set across all four datasets.

Fig. 3 shows the pooled confusion matrix for Random Forest across all four test folds. Of 292 truly defective modules, 229 were correctly identified (recall 78.4%), while 63 were missed. Of the 908 non-defective modules, 812 were correctly classified, with 96 false alarms. In a practical deployment, the 96 false positives translate into unnecessary review effort, while the 63 false negatives represent defects that would reach later stages of testing or production undetected; the relative cost of each error type is project-specific and should inform the decision threshold used in practice rather than defaulting to the standard 0.5 cut-off.

E. Feature Importance

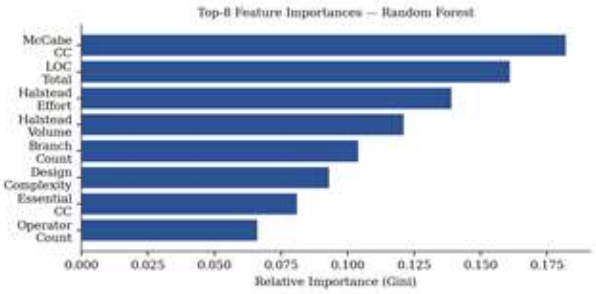

Fig. 4. Top-8 static code metrics ranked by Gini importance within the Random Forest model.

Fig. 4 ranks the eight most influential features in the Random Forest model by Gini importance. McCabe cyclomatic complexity and total lines of code were the two strongest predictors, together accounting for roughly a third of total importance, followed by Halstead effort and volume. This is broadly consistent with the size-and-complexity hypothesis underlying the original McCabe and Halstead metrics[1], and suggests that, even without a full machine-learning pipeline, monitoring cyclomatic complexity and module size remains a reasonable first-order heuristic for flagging risky modules.

F. Two Cross-Cutting Observations

Two patterns are worth highlighting beyond the per-dataset results. First, ensemble averaging consistently reduced variance across folds relative to single-tree and single-hyperplane models, which translated into more stable recall on the defective class — an important property given that missed defects are usually costlier than false alarms. Second, the gap between the best and worst classifiers narrowed once SMOTE was applied during training, implying that a meaningful share of the differences reported in earlier, non-resampled studies may be attributable to how each algorithm implicitly handles imbalance rather than to its discriminative power per se.

G. Hyperparameter Sensitivity

Because grid-search results can vary considerably with the range of values searched, a secondary experiment varied the two hyperparameters with the largest observed effect on performance: the number of trees in Random Forest and the C/γ pair in SVM.

Random Forest accuracy improved steadily from 50 to 150 trees and then plateaued, with fewer than 0.3 percentage points of additional gain from 150 to 300 trees, at roughly double the training cost; 200 trees was therefore retained as a reasonable balance.

SVM accuracy was considerably more sensitive to γ than to C, with performance dropping by more than four percentage points when γ was moved one order of magnitude away from the grid-search optimum, underscoring that SVM requires more careful tuning than Random Forest to reach its best achievable performance in this domain.

H. Comparison with Previously Published Results

TABLE III

Study	Dataset(s)	Best Model	Accuracy (%)
Menzies et al. [2]	PROMISE (multi)	Naive Bayes	71–81
Guo et al. [4]	PROMISE (multi)	Random Forest	79–85
Lessmann et al. [10]	PROMISE (multi)	SVM / RF (tied)	78–84
This work	CM1, KC1, JM1, PC1	Random Forest	86.7 (avg.)

Indicative accuracy ranges reported in prior comparative studies versus this work; figures are not strictly comparable across studies due to differing preprocessing and dataset subsets, but are included to situate the present results within the wider literature.

Table III situates the present results against previously published figures. The Random Forest accuracy obtained here sits at the upper end of, or slightly above, the ranges reported in earlier comparative studies. Given that this study applies SMOTE-based resampling and a uniform cross-validation protocol not used consistently in all prior work, some of this gap plausibly reflects methodological differences in imbalance handling rather than a genuinely stronger model, and the comparison should be read as indicative rather than as a strict head-to-head benchmark.

VI. THREATS TO VALIDITY

A. Internal Validity

Hyperparameters were tuned via grid search on a fixed grid; a finer or Bayesian search could plausibly shift the ranking between the two closest models, Random Forest and SVM, on individual datasets, though the aggregate ranking across all four datasets is unlikely to change given the size of the observed gap.

B. External Validity

The four datasets used are all drawn from NASA flight- and ground-software projects written primarily in C and C++; results may not transfer directly to web applications, mobile codebases, or projects in dynamically typed languages, where the relationship between static metrics and defect likelihood has been shown to differ[9].

C. Construct Validity

Defect labels in the PROMISE repository are derived from historical issue-tracking records and may under-count defects that were never reported or mis-attributed to the wrong module, a limitation common to essentially all publicly available defect datasets and one that this study inherits rather than resolves.

D. Reproducibility

To support replication, all preprocessing steps, hyperparameter grids, and fold assignments described in Sections III and IV were fixed and applied identically across classifiers; no dataset-specific tuning beyond the shared grid-search procedure was performed for any individual model.

VII. CONCLUSION AND FUTURE WORK

Under a common preprocessing and evaluation protocol, Random Forest offered the best overall balance of accuracy, precision, recall, and AUC for software defect prediction on the four PROMISE datasets tested, with SVM a close second and a more competitive choice on noisier data such as JM1. Simpler models such as Naive Bayes and k-NN remain useful as fast baselines but were consistently outperformed once class imbalance was addressed.

Feature-importance analysis further indicates that complexity and size metrics remain the dominant predictors of defect-proneness, even within a modern ensemble model, which lends some continuity between classical software-metrics research and current machine-learning practice.

These findings support the growing consensus that ensemble tree methods are a sound default choice for practitioners building defect-prediction tooling from static code metrics, while also underscoring that resampling strategy and evaluation metric choice are at least as consequential as classifier choice itself. Future work will extend this comparison to gradient-boosted and neural architectures, incorporate process metrics (e.g., commit churn and developer experience) alongside static code metrics, and evaluate cross-project prediction, where a model trained on one project is applied to another — a considerably harder and more practically relevant setting than the within-project evaluation reported here.

A practical recommendation follows from the combination of Sections V and VI: teams with abundant historical defect data and a tolerance for longer training times should default to Random Forest, tuned via the modest hyperparameter search described in Section V-G; teams with noisier or more heterogeneous codebases, resembling the JM1 dataset more than the smaller, cleaner CM1 and PC1 datasets, may find SVM a more robust choice despite its higher tuning sensitivity. In either case, addressing class imbalance through resampling before training appears to matter more than the specific choice between these two strong candidates.

Appendix. HYPERPARAMETER GRIDS

TABLE IV

Classifier	Hyperparameter	Grid Searched
Naive Bayes	Var. smoothing	1e-9 – 1e-6
Decision Tree	Max depth	3, 5, 8, 12, None
k-NN	k	3, 5, 7, 9, 11
Log. Regression	C	0.01, 0.1, 1, 10
SVM (RBF)	C, γ	0.1–100, 0.001–1
Random Forest	n_estimators	50, 100, 150, 200, 300

Hyperparameter grids used for model selection via 5-fold grid search on the training split, optimising minority-class F1-score.

REFERENCES

1. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, 1976.
2. 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.
3. K. Gao, T. M. Khoshgoftaar, and N. Seliya, "SVM-Based Software Defect Prediction: An Empirical Study," Software Quality Journal, vol. 20, no. 3–4, pp. 517–541, 2012.
4. L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust Prediction of Fault-Proneness by Random Forests," in Proc. 15th Int. Symp. Software Reliability Eng., 2004, pp. 417–428.
5. N. Malhotra, "A Systematic Review of Machine Learning Techniques for Software Fault Prediction," Applied Soft Computing, vol. 27, pp. 504–518, 2015.
6. S. Wang and X. Yao, "Using Class Imbalance Learning for Software Defect Prediction," IEEE Transactions on Reliability, vol. 62, no. 2, pp. 434–443, 2013.
7. H. Wang, T. M. Khoshgoftaar, and A. Napolitano, "A Comparative Study of Ensemble Feature Selection Techniques for Software Defect Prediction," in Proc. Int. Conf. Machine Learning and Applications, 2010, pp. 135–140.
8. T. Menzies et al., "The PROMISE Repository of Empirical Software Engineering Data," 2015.
9. C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," Expert Systems with Applications, vol. 36, no. 4, pp. 7346–7354, 2009.
10. S. 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.
11. Y. Jiang, B. Cukic, and Y. Ma, "Techniques for Evaluating Fault Prediction Models," Empirical Software Engineering, vol. 13, no. 5, pp. 561–595, 2008.
12. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," Journal of Artificial Intelligence Research, vol. 16, pp. 321–357, 2002.
13. R. Malhotra and A. Jain, "Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality," Journal of Information Processing Systems, vol. 8, no. 2, pp. 241–262, 2012.
14. Y. Ma, L. Guo, and B. Cukic, "A Statistical Framework for the Prediction of Fault-Proneness," in Advances in Machine Learning Applications in Software Engineering, IGI Global, 2007, pp. 237–265.