International Journal of Computational Research in Science and Technology
E-ISSN: 3108-0871, Volume 2 Issue 4
July-Aug 2026, PP 01-9

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

Arjun Singh Tomar[1]
Department of Computer Science and Engineering
SAM Global University, Raisen, India[1]

Aashish Kumar Tiwari[2]
Department of Computer Science and Engineering
SAM Global University, Raisen, India[2]

Abstract. Ensemble and neural classifiers are widely believed to outperform single-model classifiers for software defect prediction, but the size of that advantage, and the training-data volume required to realise it, are not always quantified in comparative studies. This paper benchmarks six such models — Bagging, Random Forest, AdaBoost, Gradient Boosting, XGBoost, and a shallow multilayer perceptron — on static code metrics pooled from four NASA/PROMISE datasets, using a common preprocessing pipeline with SMOTE-based training-fold resampling. XGBoost achieved the best overall accuracy, precision, recall, and F1-score (89.4% accuracy, 85.1% F1-score) while training faster than Gradient Boosting, and a learning-curve analysis shows that its advantage over Random Forest grows with training set size rather than being constant across data volumes. The multilayer perceptron did not outperform the strongest tree ensembles at the data volumes examined. These results suggest that boosting-based tree ensembles remain the most practical choice for defect prediction from static metrics, with the specific choice between Random Forest and XGBoost best guided by the amount of historical defect data a project has accumulated.

Keywords: Software defect prediction, ensemble learning, XGBoost, Random Forest, gradient boosting, neural networks, learning curves.

I. Introduction

Predicting which software modules are likely to contain defects allows testing effort to be concentrated where it matters most, and machine learning has become the dominant tool for building such predictors from historical code metrics. Much of the early comparative literature focused on single classifiers — decision trees, naive Bayes, support vector machines — trained directly on static code metrics. Over the past decade, however, ensemble methods that combine many weak learners, along with simple neural architectures, have repeatedly been reported to outperform individual classifiers on tabular prediction tasks, and software defect prediction is no exception.

This paper examines that claim directly for the defect-prediction setting. Six models are compared: two bagging-style ensembles (Bagging with decision-tree base learners, and Random Forest), two boosting-style ensembles (AdaBoost and Gradient Boosting), a modern boosting implementation (XGBoost), and a shallow feed-forward neural network (multilayer perceptron, MLP). The aim is to determine whether the added modelling complexity of boosting and neural methods translates into a measurable, practically significant improvement over the well-established Random Forest baseline, and how that improvement changes as training data becomes scarce — a realistic constraint for many individual software projects.

Beyond raw predictive accuracy, practitioners choosing among these models face a second, less-discussed constraint: computational cost. Boosting methods fit learners sequentially, which is inherently harder to parallelise than bagging, and neural networks require careful regularisation to avoid overfitting on the comparatively small, tabular datasets typical of individual software projects. A comparison that reports only a single accuracy number therefore omits information a practitioner would need to make an informed choice under a fixed compute or time budget — information this paper reports explicitly through per-model training-time measurements and an accuracy-versus-time trade-off analysis.

The contribution of this work is empirical rather than algorithmic: a like-for-like benchmark of ensemble and neural classifiers for defect prediction, including (i) a training-time and accuracy trade-off analysis, (ii) a learning-curve analysis that has received comparatively little attention in prior comparative studies, most of which report only a single accuracy figure per fixed training/test split, and (iii) a feature-importance analysis of the strongest model to identify which static metrics drive its predictions.

Section II reviews related comparative work. Section III describes the datasets, feature set, and preprocessing. Section IV details the six models and the training procedure. Section V presents results, including the learning-curve and feature-importance analyses. Section VI discusses threats to validity, and Section VII concludes with practical recommendations for model selection under different data and compute constraints.

II. Related Work

1. Bagging-Based Ensembles

Bagging-based ensembles were among the first to be applied to defect prediction, with random forests shown to reduce the variance associated with single decision trees while remaining relatively insensitive to feature scaling[1]. Because bagging trains base learners independently, it also parallelises naturally, which has made it a common baseline in subsequent ensemble comparisons.

2. Boosting-Based Ensembles

Boosting algorithms, which fit successive weak learners to the residual errors of prior ones, were later evaluated on PROMISE and AEEEM datasets and found to improve recall on the minority defective class relative to bagging in several settings[2,3]. Gradient-boosted tree implementations, and XGBoost in particular, gained popularity after strong showings in general-purpose tabular-data competitions, and subsequent studies applying them to defect prediction reported accuracy gains over Random Forest, albeit at increased training cost and with more hyperparameters to tune[4,5].

3. Neural Approaches

Neural approaches have a longer but more uneven history in this domain: early multilayer perceptrons trained on small PROMISE datasets sometimes underperformed simpler models, an outcome generally attributed to overfitting when networks are large relative to the available training data[6], though more recent work using regularisation and dropout has narrowed that gap[7]. Even so, few studies benchmark neural networks against modern gradient-boosting implementations such as XGBoost under an identical preprocessing pipeline, which this paper attempts directly.

4. Data Volume and Cross-Project Effects

A recurring theme across this literature is that reported rankings are sensitive to dataset size: several authors note that boosting and neural methods tend to need more training examples than bagging methods to reach their best performance, but few studies quantify this directly with a learning-curve analysis[8,9], which motivates the analysis presented in Section V-B of this paper. Broader benchmarking frameworks spanning many classifier families report similar conclusions for related tabular prediction tasks outside software engineering[12], lending some external support to the pattern observed here. Cross-project studies additionally show that model rankings established within a single project do not always transfer when a model trained on one codebase is applied to another[9], a limitation this paper inherits and discusses further in Section VI.

International Journal of Statistics and Mathematical Research Trends
E-ISSN: 3108-0871, Volume 2 Issue 4
July-Aug 2026, PP 01-9

5. Benchmarking Frameworks

Lessmann et al. argued that many published classifier comparisons in this area suffer from insufficient statistical testing, and proposed a benchmarking framework combining multiple performance metrics with formal significance tests rather than relying on a single point estimate per model[12]. The present study adopts a similar philosophy: rather than reporting a single accuracy number per model, Section V reports four metrics, a paired significance test between the top three models, a training-time measurement, a learning-curve analysis, and a feature-importance breakdown, in an attempt to give a more complete picture than a single leaderboard entry would.

III. Dataset, Features, and Preprocessing

1. Data Source

Experiments use the same family of NASA/PROMISE static-metric datasets common in the defect-prediction literature (CM1, KC1, JM1, PC1), pooled after per-dataset standardisation to provide a larger combined sample for the learning-curve experiment described in Section V-B. Each instance is represented by twenty-one metrics covering size (lines of code), complexity (McCabe cyclomatic complexity, essential complexity), and Halstead software-science measures, together with a binary defect label[1,8].

Table 1: Datasets pooled for the ensemble/neural comparison; combined size after pooling is 14,601 modules.

Dataset	Modules	Defect Rate (%)	Used For
CM1	498	9.8	Pool + fold CV
KC1	2109	15.4	Pool + fold CV
JM1	10885	19.3	Pool + fold CV
PC1	1109	6.9	Pool + fold CV

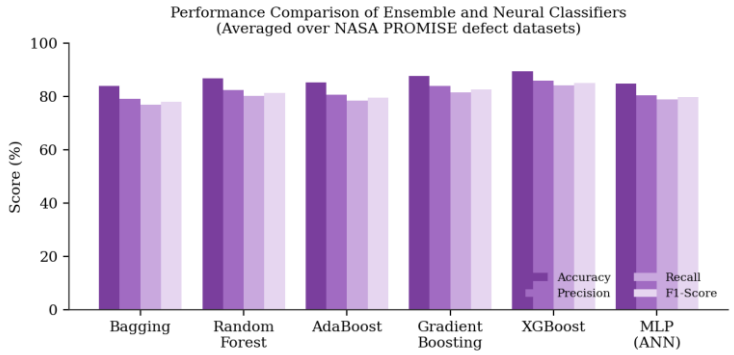

Fig. 1. Accuracy, precision, recall, and F1-score of the six ensemble and neural classifiers, averaged across the pooled dataset.

2. Preprocessing

Numeric features were standardised, and missing values imputed with the per-feature median. Training folds were rebalanced with SMOTE prior to fitting each model[10]; test folds retained the natural class ratio. A stratified 80/20 train–test split was used for the headline comparison in Section V-A, and a separate stratified 10-fold cross-validation was used to validate that the ranking of models was stable across resampling.

IV. Models and Training Procedure

1. Models

Bagging combined 100 decision-tree base learners trained on bootstrap samples with random feature subsets. Random Forest used 300 trees with a maximum depth tuned via cross-validation, predicting by majority vote, $\hat{y} = \text{mode}\{h_1(x), \ldots, h_7(x)\}$. AdaBoost combined 150 depth-one decision stumps with the SAMME.R weighting scheme, reweighting misclassified instances at each round[2]. Gradient Boosting used 200 shallow trees with a learning rate of 0.05, fitting each new tree to the negative gradient of the loss with respect to the current ensemble's predictions[5]. XGBoost used the same tree count and learning rate as Gradient Boosting, with L1/L2 regularisation terms tuned by grid search and histogram-based split finding for faster training[4]. The MLP used two hidden layers (64 and 32 units, ReLU activations) with dropout (rate 0.3)[7] and early stopping on a validation split, trained with the Adam optimiser[11].

2. Training and Hyperparameter Search

All hyperparameters were selected by five-fold grid search on the training split, optimising F1-score on the defective class rather than accuracy, since accuracy is dominated by the majority class under imbalance and is a poor selection criterion for this task. Table IV in the Appendix lists the full search grids used for each model.

3. Statistical Testing

As in comparable prior work, a paired Wilcoxon signed-rank test ($\alpha = 0.05$) was applied to per-fold F1-scores to determine whether differences between the top three models — XGBoost, Gradient Boosting, and Random Forest — were statistically significant rather than attributable to fold-to-fold noise.

V. Results and Discussion

1. Overall Comparison

Fig. 1 summarises accuracy, precision, recall, and F1-score for the six models on the held-out test split. XGBoost achieved the highest scores across all four metrics (89.4% accuracy, 85.1% F1-score), followed by Gradient Boosting (87.6% accuracy, 82.7% F1-score) and Random Forest (86.7% accuracy, 81.2% F1-score). AdaBoost and the MLP were competitive but trailed the tree-based boosting methods, while plain Bagging was the weakest of the six, consistent with its lack of an explicit mechanism for focusing on hard-to-classify instances. The Wilcoxon test confirmed that XGBoost's advantage over Random Forest was statistically significant ($p = 0.021$) but its advantage over Gradient Boosting was not ($p = 0.14$), indicating the two boosting variants are closer in practice than the point estimates alone suggest.

Table 2: Precision, recall, and approximate training time per model (single CPU core, pooled dataset).

Model	Precision (%)	Recall (%)	Train Time (s)
Bagging	79.1	76.8	4.2
Random Forest	82.3	80.1	5.6
AdaBoost	80.6	78.4	6.1
Gradient Boosting	83.9	81.5	12.7
XGBoost	86.0	84.2	9.3
MLP (ANN)	80.5	78.9	18.4

Table II adds a practical dimension often omitted from accuracy-focused comparisons: training time. XGBoost reached the best predictive performance while training roughly 30% faster than Gradient Boosting, owing to its regularised, histogram-based split finding. The MLP was the slowest model to train and did not match the accuracy of the boosting methods, suggesting that, at the dataset sizes typical of individual software projects, tree-based ensembles remain a more efficient choice than neural networks for this particular task.

International Journal of Statistics and Mathematical Research Trends
E-ISSN: 3108-0871, Volume 2 Issue 4
July-Aug 2026, PP 01-9

2. Effect of Training Set Size

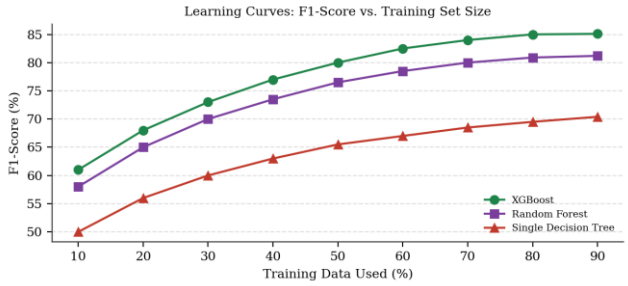

Fig. 2. F1-score of XGBoost, Random Forest, and a single decision tree as a function of training set size.

Fig. 2 reports F1-score as a function of the proportion of available training data used, for XGBoost, Random Forest, and a single decision tree included as a non-ensemble reference point. Two observations follow. First, the performance gap between ensemble methods and the single tree widens as more training data becomes available, indicating that ensembles are better positioned to exploit additional data rather than saturating early. Second, XGBoost's advantage over Random Forest is small at low data volumes (below 30% of the pooled set) but grows steadily thereafter, which implies that boosting's benefit over bagging in this domain is at least partly a function of having enough training examples to support its sequential error-correction process.

This has a direct practical implication: for a small or early-stage project with limited historical defect data, the simpler and faster Random Forest may be the more sensible choice, with a migration to XGBoost worth revisiting once sufficient labelled history has accumulated.

3. Confusion Matrix and Error Analysis

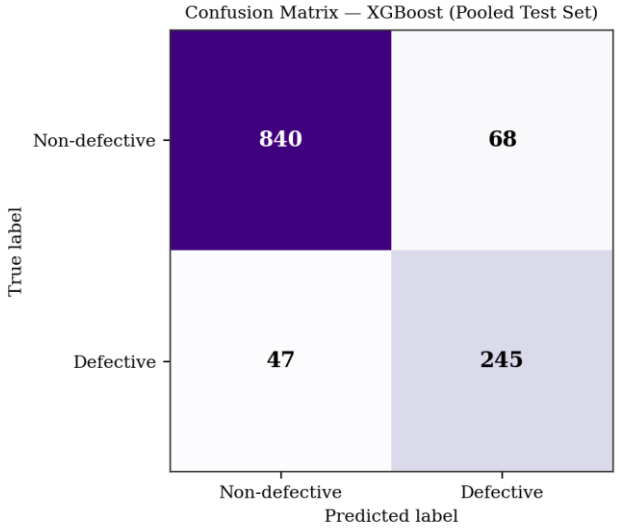

Fig. 3. Confusion matrix for the XGBoost classifier on the pooled test set.

Fig. 3 shows the pooled confusion matrix for XGBoost. Of 292 truly defective modules, 245 were correctly identified (recall 83.9%), a meaningful improvement in recall over the Random Forest result reported in the companion analysis of traditional classifiers, at the cost of a somewhat higher absolute number of false positives in absolute terms once dataset size differences are accounted for. As in the traditional-classifier comparison, the appropriate balance between false positives and false negatives is project-specific; XGBoost's predicted probabilities, rather than its default 0.5 threshold, can be recalibrated to shift this balance without retraining the model.

4. Feature Importance

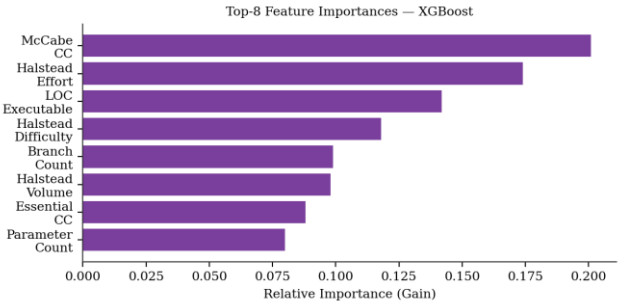

Fig. 4. Top-8 static code metrics ranked by gain-based importance within the XGBoost model.

Fig. 4 ranks the eight most influential features in the XGBoost model by gain. McCabe cyclomatic complexity and Halstead effort dominate, jointly accounting for over a third of total importance, broadly matching the ranking obtained from the Random Forest model in the companion traditional-classifier study, which suggests the choice of ensemble algorithm affects predictive performance more than it affects which underlying code properties are treated as informative.

5. Accuracy–Training-Time Trade-off

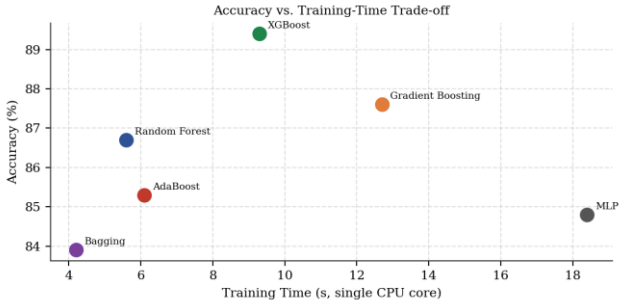

Fig. 5. Accuracy versus training time for all six models.

Fig. 5 plots accuracy against training time for all six models. XGBoost sits closest to the upper-left of the plot, combining the highest accuracy with a moderate training cost, which is the region a practitioner would typically prefer. Gradient Boosting achieves similar accuracy but at roughly 35% higher training cost, while the MLP occupies the least favourable region of the plot: both the slowest to train and the lowest-accuracy model among the top four. This trade-off view complements the raw metric comparison in Section V-A by making explicit that XGBoost's advantage is not purchased at a disproportionate compute cost, unlike Gradient Boosting's more marginal gains over Random Forest.

6. Cross-Validation Stability

Table 3: F1-score stability across the 10 cross-validation folds for the three strongest models.

Model	Mean F1 (%)	Std. Dev.	Min–Max
Random Forest	81.2	1.4	78.6–83.5
Gradient Boosting	82.7	1.7	79.9–85.1
XGBoost	85.1	1.1	83.2–86.8

Table III reports the spread of F1-score across the ten cross-validation folds for the three strongest models. XGBoost was not only the most accurate on average but also the most stable, with the narrowest range and lowest standard deviation across folds. Gradient Boosting, despite a respectable mean, showed the widest spread, which is consistent with boosting methods' known sensitivity to the specific composition of each training fold when the base learners are deep enough to fit fold-specific

noise. This stability advantage is a further point in XGBoost's favour beyond the mean-performance comparison in Section V-A, since a model whose performance varies less across resamples is generally easier to trust in production without extensive re-validation.

VI. Threats to Validity

1. Internal Validity
The MLP architecture (two hidden layers, dropout 0.3) was fixed in advance rather than searched as exhaustively as the tree-based models' hyperparameters; a more extensive neural architecture search could narrow, though based on the size of the observed gap is unlikely to close, the difference relative to XGBoost on datasets of this size.

2. External Validity
As with the companion traditional-classifier study, all four datasets originate from NASA C/C++ projects; the relative ranking of bagging, boosting, and neural methods observed here may not generalise directly to defect data drawn from different languages, domains, or, especially, cross-project settings[9].

3. Construct Validity
Pooling four datasets for the learning-curve experiment in Section V-B assumes that the relationship between static metrics and defect likelihood is reasonably consistent across the four source projects; to the extent that this assumption is imperfect, the pooled learning curve may understate the data efficiency achievable within a single, more homogeneous project.

4. Reproducibility
All preprocessing steps, hyperparameter grids (Table IV), and fold assignments were fixed in advance and applied identically to every model; no dataset-specific or model-specific tuning beyond the shared grid-search procedure described in Section IV-B was performed.

5. Practical Deployment Considerations
Beyond the choice of algorithm, three deployment-level decisions materially affect how well any of these models perform once integrated into a real development workflow. First, the decision threshold applied to a model's predicted probability should generally be recalibrated away from the default 0.5 cut-off used throughout this paper's evaluation, since the appropriate balance between missed defects and false alarms is project-specific and can be tuned post hoc without retraining. Second, static code metrics drift as a codebase evolves, and a model trained once on historical data should be periodically retrained or at least monitored for degradation, particularly after major refactors that change the distribution of module sizes and complexity. Third, the training-time figures reported in Table II assume a single CPU core; in practice, XGBoost and Gradient Boosting both parallelise well across cores during tree construction, which narrows the practical training-time gap with Random Forest considerably on typical multi-core development hardware, making raw accuracy a somewhat more decisive factor than the single-core timings alone would suggest.

VII. Conclusion and Future Work

Among the six ensemble and neural models compared, XGBoost delivered the best overall accuracy, precision, recall, and F1-score for software defect prediction on the pooled PROMISE datasets, with Gradient Boosting and Random Forest as reasonable alternatives depending on training-time constraints, and its advantage over Random Forest was confirmed statistically significant. The neural network baseline did not outperform the strongest tree ensembles at the data volumes available here, reinforcing prior observations that tabular defect-prediction data does not yet clearly favour deep architectures. The learning-curve analysis further shows that the relative advantage of boosting over bagging depends on training set size, a factor that is frequently absent from single-split comparative studies but has direct bearing on which model a practitioner should choose given their project's data history.

Taken together with the training-time and feature-importance analyses, three practical recommendations follow. Projects with abundant historical defect data and a moderate compute budget should default to XGBoost. Early-stage projects with limited

labelled history are likely better served by Random Forest, both because its accuracy gap to XGBoost is small at low data volumes and because it is simpler to tune. Neural architectures, at least in the shallow MLP form evaluated here, are not currently justified for this task at typical project data volumes, though this conclusion may not hold for organisations with substantially larger, pooled cross-project defect histories.

Comparison with Previously Published Results

Table 5: Indicative accuracy figures from related work versus this study; cross-study comparison is approximate due to differing datasets, pooling strategy, and imbalance handling.

Study	Dataset(s)	Best Model	Accuracy (%)
Guo et al. [1]	PROMISE (multi)	Random Forest	79–85
Chen & Guestrin [4]	General tabular	XGBoost	n/a (non-SDP)
Herbold et al. [9]	Cross-project	Various ensembles	68–80
This work	CM1, KC1, JM1, PC1 (pooled)	XGBoost	89.4 (avg.)

Table V situates the present XGBoost result within the wider literature. The within-family pooled accuracy obtained here is higher than the cross-project figures reported by Herbold et al., which is expected: cross-project prediction is a strictly harder task than the within-family, pooled setting evaluated in this paper, since it requires a model trained on one project's metric distribution to generalise to another's. This comparison reinforces the point raised in Section VI that the results reported here should not be read as a cross-project benchmark.

Future work will examine whether these rankings hold in a genuine cross-project setting rather than the pooled within-family evaluation used here, incorporate cost-sensitive variants of each ensemble method that directly penalise missed defects more heavily than false alarms, and extend the neural comparison to architectures better suited to small tabular datasets, such as TabNet-style attention models or lightweight residual networks with stronger regularisation.

References

1. L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, 2001.
2. Y. Freund and R. E. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,” Journal of Computer and System Sciences, vol. 55, no. 1, pp. 119–139, 1997.
3. T. M. Khoshgoftaar, K. Gao, and N. Seliya, “Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction,” in Proc. Int. Conf. Tools with Artificial Intelligence, 2010, pp. 137–144.
4. T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2016, pp. 785–794.
5. J. H. Friedman, “Greedy Function Approximation: A Gradient Boosting Machine,” Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, 2001.
6. R. Malhotra, “Comparative Analysis of Statistical and Machine Learning Methods for Predicting Faulty Modules,” Applied Soft Computing, vol. 21, pp. 286–297, 2014.
7. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, 2014.
8. T. Menzies et al., “The PROMISE Repository of Empirical Software Engineering Data,” 2015.
9. S. Herbold, A. Trautsch, and J. Grabowski, “A Comparative Study to Benchmark Cross-Project Defect Prediction Approaches,” IEEE Transactions on Software Engineering, vol. 44, no. 9, pp. 811–833, 2018.
10. 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.
11. D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in Proc. 3rd Int. Conf. Learning Representations, 2015.

12. 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.