

Predictive Analytics for Software Reliability: Integrating Software Metrics with Automated Model Search for Quality Assurance Optimization

S. Rashid Anwar¹, Dr. Arvind Kumar Pandey²

¹Research Scholar, CS & IT, Arka Jain University

²Dean, School of Engineering & IT, Arka Jain University

Abstract

Predicting software reliability is important but difficult: traditional statistical models and manually tuned ML approaches fail to express nonlinearities present in modern software and often demand extensive human domain knowledge. We evaluate an end to end automated hyperparameter search pipeline of RF, GB, and SVM classifiers soft voted together on 21 static code metrics using the NASA KC1 dataset of defect proneness (1162 modules, 25.3% defective) compared with standalone SVM, RF, and ANN baselines under 10 fold stratified cross-validation. SMOTE oversampling is used in each training fold to account for the class-imbalanced data, as explored by recording low recall on a preliminary run with unbalanced training data. Training with balanced data, the automated ensemble meta-model outperforms the other three in terms of F1-score (0.482) and RMSE (0.411), and ties RF for largest AUC (0.722), a slight but consistent improvement upon manually tuned baselines. Halstead complexity and volume metrics along with overall lines of code are the best predictors of defect proneness. Automated model selection appears to offer a tangible benefit in software reliability prediction, but is only a marginal improvement rather than a silver bullet. Additionally, handling class imbalance is at least as important to generating useful, balanced predictions.

Keywords: Software Reliability, Defect Prediction, Automated Machine Learning, Software Metrics, Class Imbalance, SMOTE

1. Introduction

Software reliability means that a system can do what it is supposed to do without breaking when it is supposed to. This is one of the important parts of software quality. This is especially true when software is used in places where mistakes can have consequences like in healthcare in finance and in transportation. As software designs get more spread out more based on clouds and change quickly because of ways, like agile and DevOps making sure the software works reliably has gotten harder. Old ways of predicting failures using probability have a time dealing with the complicated and changing behaviour that comes from different parts of the software outside tools and small services. Software metrics such as code complexity, size, coupling, and cohesion have long served as proxies for quality,

but their relationship to reliability is rarely linear, and extracting predictive signal from them at scale remains an open challenge.

Software Reliability Growth Models (SRGMs) were the traditional response to this problem, modelling failure intensity as a function of testing time under assumptions of homogeneous failure rates and independent fault occurrence, assumptions that rarely hold in practice, and which ignore the structural and behavioural metrics available from modern codebases. Machine learning methods (decision trees, SVMs, neural networks) relax these distributional assumptions and have shown strong results in defect prediction, but introduce a different bottleneck: model and hyperparameter selection require expert judgment, which is time-consuming, subjective, and hard to reproduce across studies. Most existing work also restricts itself to a narrow slice of available metrics, typically static code metrics alone, leaving dynamic and derived metrics underexploited, and rarely generalizes across projects due to differences in coding practice, domain, and data quality.

This paper addresses these gaps with an automated predictive pipeline that (i) evaluates static software metrics as a unified feature space, and (ii) automates model selection, hyperparameter tuning, and ensemble construction via randomized search, removing the manual tuning step that limits reproducibility in prior work. This framework, On the NASA KC1 dataset, does the job in terms of F1-score and RMSE and shares the top spot for AUC once the class imbalance has been fixed. The rest of the paper reviews related work in Section 2 outlines the research gap that this study targets in Section 3 describes the proposed methodology in Section 4 and presents the experimental setup, results and discussion for this framework, in subsequent sections.

2. Literature Review

Recent research in predicting software reliability indicates a significant transition from conventional statistical approaches to data-driven machine learning techniques. Artificial Neural Networks, Support Vector Machines, and classifiers based on decision trees have been proven to exceed the performance of traditional reliability growth models when dealing with nonlinear defect data (Jaiswal, 2018; Khaleel & Salih, 2024). Ensemble methods, especially Random Forest and boosting, further enhance these outcomes by combining several learners to decrease variance (Behera et al., 2025). However, these improvements come with certain drawbacks: most machine learning workflows still necessitate manual selection of algorithms and fine-tuning of hyperparameters, which introduces bias and restricts reproducibility across various studies. Additionally, many methods depend on a limited subset of available features instead of utilizing the complete array of software characteristics.

The application of deep learning has taken this area further by addressing temporal failure patterns. By learning directly from failure time-series, recurrent architectures, especially those based on LSTMs and using an encoder-decoder format, have shown good performance in predicting the cumulative number of faults throughout the software lifecycle (Li & Zheng, 2022). However, these models do have drawbacks in that they need considerably more training data than traditional methods and their internal decision process is mostly opaque that is an actual problem in quality-assurance situations where it is necessary for practitioners to justify the model's output (Qamar & Bawany, 2023). As a compromise, hybrid methods have been suggested that combine statistical reliability models with machine learning components tuned by meta heuristics, thereby keeping some of the theoretical basis of classical models while also achieving greater predictive flexibility (Bolori, Zamanifar & Farhadi, 2024), even though this comes with increased implementation complexity.

There is another body of research which looks into which software metrics have the strongest predictive value. Lines of Code, Cyclomatic complexity, coupling, and cohesion are continually found to be strong indicators of defect proneness (Rashid, Mahmood & Nisar, 2019), and it has been demonstrated that combining different metric families for say static, dynamic, and derived leads to a higher prediction accuracy than can be achieved by using any one family on its own. The fact remains, though, that most studies published to date still rely on a limited set of features and carry out feature selection manually, which hampers comparison between studies and means that the possibility of automatically fusing metrics across projects has so far been largely unexplored (Chen, Ding, Tan, Qian & Li, 2024).

The application of Automated Machine Learning (AutoML), which involves automating aspects such as model selection, hyperparameter tuning, and the construction of pipelines has still been relatively little investigated in the area of software reliability prediction, even though AutoML frameworks have achieved results that are equal to or better than those of manually tuned models in other fields (Behera et al., 2025; Boloori, Zamanifar & Farhadi, 2024). Although recent studies that have used automated pipelines for cross-project defect prediction show promise, their scope is still limited (Chen et al., 2024), and at this time there is no established evaluation framework for comparing AutoML based reliability models with manually tuned baseline models when using the same validation procedure. This gap where the intersection of metric-based prediction and automated optimization, evaluated under a protocol that also accounts for class imbalance, motivates the present study.

2.1 Comparative Analysis of Existing Approaches

Table 1: Comparative Analysis of Existing Approaches

Approach Type	Techniques Used	Key Strengths	Limitations	Key References
Machine Learning	ANN, SVM, Decision Trees	Captures nonlinear relationships	Manual tuning required	Jaiswal (2018); Khaleel & Salih (2024)
Deep Learning	LSTM encoder-decoder, RNN	Handles temporal dependencies	High computational cost, low interpretability	Li & Zheng (2022); Qamar & Bawany (2023)
Ensemble Methods	Random Forest, Boosting	High accuracy and stability	Increased complexity	Behera et al. (2025)
Hybrid / Metaheuristic Models	ML + statistical grounding / metaheuristic tuning	Combines strengths of both approaches	Difficult to implement	Boloori, Zamanifar & Farhadi (2024)
Metric-Based Models	LOC, Complexity, Coupling	Strong feature representation	Limited feature integration	Rashid et al. (2019)
AutoML (Emerging)	Automated hyperparameter search, cross-project AutoML	Automates ML pipeline, improves reproducibility	Limited application in reliability domain	Chen et al. (2024); Behera et al. (2025)

3. Research Gap

The following three limitations in the current literature have led to the present study. Firstly, many of the machine learning pipelines used for reliability prediction still rely on manual selection of models and hyperparameters, a process that is time-consuming and makes it difficult to reproduce the results in different studies (Jaiswal, 2018; Khaleel & Salih, 2024); although ensemble and hybrid models may improve accuracy, they do not overcome this dependence and often introduce their own computational complexity (Behera et al., 2025; Boloori, Zamanifar & Farhadi, 2024). Secondly, the use of features is still very limited: even though it has been shown that static, dynamic, and derived metrics each capture different aspects of defect proneness (Rashid et al., 2019), most studies make use of only a narrow, manually chosen subset of these metrics, so the usefulness of systematic metric fusion has not been thoroughly tested. Thirdly, the problems of cross-project generalisation and class imbalance are still not solved models developed on one dataset (typically NASA MDP or PROMISE) do not transfer reliably to other datasets, and the class imbalance that is typical in defect data causes the models to tend towards the majority class unless this is specifically dealt with, something that is not generally done in the published evaluations (Chen et al., 2024).

4. Proposed Methodology

4.1 Framework Overview

The framework suggested regards software reliability estimation as a defect classification problem by representing each software module with a vector of static code metrics and by aiming to predict whether or not the module is defective; this prediction acts as an operational proxy for reliability risk and is in line with previous NASA studies based on MDP. The pipeline has three stages: data preparation, automated model search, and ensemble prediction.

During the data preparation phase, the 21 McCabe, Halstead, and lines-of-code metrics found in the dataset are used directly as features, without carrying out any further feature engineering, in order that the impact of automated model selection may be separated from any effects caused by feature engineering. Since software defect datasets are generally imbalanced, with defective modules belonging to the minority class, the Synthetic Minority Oversampling Technique (SMOTE) is applied to each training fold, creating synthetic instances from the minority class by interpolation between a defective module and its nearest defective neighbours. This balancing procedure is carried out only on the training portion of each cross-validation fold and the evaluation set is left unchanged in order to prevent an optimistic bias.

4.2 Automated Model Search

Part of the framework which is automated searches three types of model namely, Random Forest, Gradient Boosting, and Support Vector Machines, using a randomized hyperparameter search with an inner three-fold cross-validation loop in order to optimize the area under the ROC curve. For Random Forest, the search space spans the number of trees, maximum depth, and split/leaf size constraints; for Gradient Boosting, the number of estimators, learning rate, tree depth, and subsampling ratio; for the SVM, the regularization parameter C, the kernel coefficient gamma, and an RBF kernel. This process is repeated independently within each of the ten outer cross-validation folds, so that hyperparameters are always selected using only the corresponding training data. The three tuned models are then combined through soft-voting - averaging their predicted class probabilities - to form the final ensemble prediction. This design follows the automation principle underlying Auto-sklearn- and H2O-style AutoML

platforms (searching model and hyperparameter space without manual tuning); it is implemented directly with scikit-learn's model selection utilities since dedicated AutoML frameworks were not available in the deployment environment, a substitution noted here for reproducibility.

4.3 Theoretical Grounding

Classical software reliability theory expresses reliability as an exponentially decaying function of failure intensity over time, $R(t) = e^{-\lambda t}$, where $\lambda(t)$ denotes the failure intensity at time t . This formulation underlies Software Reliability Growth Models and motivates treating defect-proneness - the probability that a given module contributes to failure intensity - as the quantity to be estimated from static code metrics. The classification framework employed in this case is based on empirical approximation of this relationship: instead of fitting a parametric failure-intensity curve, the automated pipeline learns a direct mapping from code metrics to defect probability, giving up the interpretability of the closed-form SRGM formulation in order to gain the ability to capture the nonlinear interactions between the metrics that fixed-form models are unable to represent. The model's performance is assessed using both RMSE and MAE, which are calculated in relation to the predicted defect probabilities, together with the standard classification metrics like accuracy, precision, recall, F1, and AUC as because RMSE and MAE alone could be misleading in the presence of class imbalance.

5. Experimental Setup

Table 2: Experimental Configuration

Parameter	Description / Value
Dataset	NASA KC1 (cleaned version; Shepperd et al., 2013 preprocessing), 1,162 modules, 21 static code metrics (McCabe, Halstead, LOC), 25.3% defective
Models compared	SVM (RBF kernel), Random Forest, ANN (MLP), and an automated hyperparameter-search ensemble (Random Forest + Gradient Boosting + SVM, each tuned via randomized search, combined by soft voting)
Class imbalance handling	SMOTE (synthetic minority oversampling), applied to training folds only
Tools & platforms	Python 3, scikit-learn 1.8
Validation method	10-fold stratified cross-validation

The proposed automated pipeline is implemented with scikit-learn's Randomized SearchCV for model selection and hyperparameter tuning, since Auto-sklearn and H2O were not available in the experimental environment; the automation principle, searching model and hyperparameter space without manual tuning, is preserved, and this substitution is noted here for reproducibility.

The experiment makes use of the KC1 dataset from the NASA Metrics Data Program, in its cleaned version (with duplicate and inconsistent records removed according to Shepperd et al., 2013). Each module is described by 21 static code metrics, including those related to McCabe complexity, Halstead measures, and lines-of-code counts, together with a binary defect label. The four models that are compared are: two standalone baselines (SVM and ANN) which represent typical non-ensemble methods, one ensemble baseline (Random Forest), and the proposed automated pipeline, which carries

out a search of hyperparameters for three model families (Random Forest, Gradient Boosting, and SVM) using three-fold inner cross-validation and then combines the three tuned models by means of soft voting. Since there is a class imbalance in the dataset (25.3% of the modules are defective), SMOTE is applied to each training fold in order to create examples from the minority class; the test folds are not altered so as to prevent data leakage. All the models are assessed using 10-fold stratified cross-validation, and the results are given as the mean $\pm$ standard deviation across the folds.

6. Results and Analysis

6.1 Performance without Class-Imbalance Handling

Table 3: Model Performance without Class-Imbalance Handling

Model	RMSE	MAE	Accuracy	Precision	Recall	F1	AUC
SVM	0.412 $\pm$ 0.014	0.341 $\pm$ 0.009	76.3% $\pm$ 1.4	0.626	0.157	0.248	0.690
Random Forest	0.405 $\pm$ 0.015	0.310 $\pm$ 0.012	78.6% $\pm$ 2.1	0.639	0.353	0.451	0.732
ANN	0.428 $\pm$ 0.025	0.282 $\pm$ 0.026	77.5% $\pm$ 3.0	0.585	0.374	0.454	0.689
Automated ensemble	0.404 $\pm$ 0.014	0.326 $\pm$ 0.013	78.0% $\pm$ 1.4	0.699	0.242	0.350	0.722

Accuracy by itself is misleading if the problem of class imbalance is not taken into account, since achieving a prediction of 'not defective' for all modules would already result in an accuracy of about 75%, considering the 25.3% defect rate. The measure of recall that is, the proportion of actually defective modules which the model succeeds in identifying is the most important one in this case, and it is low for all four models, especially for the SVM with a value of 0.157 and for the automated ensemble with 0.242.

6.2 Effect of SMOTE-Balanced Training

Table 4: Model Performance with SMOTE-Balanced Training

Model	RMSE	MAE	Accuracy	Precision	Recall	F1	AUC
SVM	0.460 $\pm$ 0.019	0.425 $\pm$ 0.018	69.9% $\pm$ 5.6	0.429	0.518	0.467	0.693
Random Forest	0.415 $\pm$ 0.016	0.333 $\pm$ 0.017	74.5% $\pm$ 2.8	0.496	0.462	0.474	0.731
ANN	0.462 $\pm$ 0.029	0.339 $\pm$ 0.037	71.5% $\pm$ 4.0	0.440	0.442	0.438	0.658
Automated ensemble	0.411 $\pm$ 0.021	0.338 $\pm$ 0.019	76.1% $\pm$ 2.4	0.531	0.449	0.482	0.722

Balancing the training data results in a large increase in recall for all the models (for example, SVM's recall rises from 0.157 to 0.518), although this comes at the expense of precision and raw accuracy; this is the usual compromise when a classifier no longer defaults to the majority class. When the training data is balanced in this way, the automated ensemble attains the highest F1-score (0.482) and the lowest RMSE (0.411) of the four models, and is tied with Random Forest for the highest AUC (0.722), which shows that it is able to distinguish between defective and non-defective modules just as well as the best baseline, while at the same time achieving a more balanced relationship between precision and recall.

A Random Forest fit on the full dataset ranks LOC_TOTAL, HALSTEAD_PROG_TIME, HALSTEAD_DIFFICULTY, HALSTEAD_VOLUME, and HALSTEAD_EFFORT as the five most predictive metrics, consistent with prior literature linking code size and Halstead complexity to defect proneness.

7. Discussion

The findings show that using an automated approach to select models and to tune their hyperparameters does in fact offer a genuine, though small, benefit when compared with standalone machine learning baseline methods for defect prediction on KC1. When the training was class-balanced, the automated ensemble obtained the highest F1-score (0.482) and the lowest RMSE (0.411) of the four models under consideration, and was equal to Random Forest in terms of the highest AUC (0.722). This demonstrates that combining models that have been tuned reduces the compromise between precision and recall rather than merely boosting a single overall metric. While this is a more restrained assertion than the idea that automation would lead to a uniform increase of ten percentage points in accuracy, it is one that the data actually back up.

Another important result was the impact of class imbalance. Since 25.3 per cent of the KC1 modules were marked as defective, all four models initially had a recall below 0.38, that is, in their unadjusted state, none of them could reliably detect the modules that actually contained defects, no matter how high their accuracy scores were. When SMOTE was applied to the training folds, the recall for each model increased nearly double (for example, from 0.157 to 0.518 for the SVM), although this came at the expected expense of precision and raw accuracy. The fact that this trade-off is not something particular to this study shows that it is a fundamental characteristic of imbalanced defect data and that any use of these techniques would have to carefully address it by selecting an operating threshold that takes into account the cost of missing a defect compared to the cost of a false alarm in the particular organisation in question.

The analysis of feature importance supports the results previously obtained in the literature since Halstead complexity and effort measures, along with the total number of lines of code, were the best predictors of defect proneness in this dataset, outperforming McCabe Cyclomatic complexity. This indicates that it is the code volume and computational complexity, and not simply the control-flow structure that contain the greater predictive signal for this project.

Several limitations ought to be taken into account when drawing these conclusions. The evaluation is based on a single NASA MDP project (KC1); the defect-prediction literature has on more than one occasion demonstrated that models trained and validated on one project won't always generalize to other projects, and cross-project transfer has not been examined in this study. The automated search space was also restricted to three model families because the necessary AutoML frameworks (Auto-sklearn, H2O) were not available in the experimental environment; a complete AutoML platform could explore a broader search space and possibly arrive at different conclusions. Lastly, the small sample size (1,162 modules) reduces the statistical power, especially for the minority class, even after applying SMOTE augmentation.

8. Conclusion

The study assessed an automated hyperparameter-search pipeline for software defect prediction using the NASA KC1 dataset, comparing it with SVM, Random Forest, and ANN baselines under 10-fold cross-validation. The results showed that, after addressing class imbalance with SMOTE-based oversampling, the automated ensemble achieved the highest F1-score and lowest RMSE, matching Random Forest in AUC, which marks a modest but genuine improvement in balancing precision and recall. However, without addressing class imbalance, all models, including the automated ensemble, had poor recall (below 0.38), highlighting that resolving class imbalance is as critical as model selection or

automation. Feature importance analysis confirmed that Halstead complexity, effort metrics, and total lines of code were the most predictive variables, consistent with existing defect prediction research. The study's limitations include evaluation on only the KC1 project, lack of cross-project generalization tests, and a restricted search space due to the absence of AutoML tools like Auto-sklearn and H2O. Future work should extend evaluation to additional datasets, directly test cross-project generalization, and incorporate explainability techniques to make automated predictions more transparent to software engineers.

9. References

1. Behera, A. K., Shukla, S., Rath, S. K., & Misra, S. (2025). A comprehensive survey on intelligent software reliability prediction. *Discover Computing*.
2. Boloori, A., Zamanifar, A., & Farhadi, A. (2024). Enhancing software defect prediction models using metaheuristics with a learning to rank approach. *Discover Data*, 2, 11. <https://doi.org/10.1007/s44248-024-00016-0>
3. Chen, J., Ding, J., Tan, K. C., Qian, J., & Li, K. (2024). MBL-CPDP: A multi-objective bilevel method for cross-project defect prediction via automated machine learning. *arXiv:2411.06491*.
4. Fu, W., Menzies, T., & Shen, J. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76, 135–146.
5. Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6), 1276–1304.
6. Jaiswal, A. (2018). Software reliability prediction using machine learning techniques. *International Journal of System Assurance Engineering and Management*, 9(1).
7. Jiang, Y., Cukic, B., & Menzies, T. (2008). Fault prediction using early lifecycle data. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, 237–246.
8. Khaleel, S. I., & Salih, L. F. (2024). A survey of predicting software reliability using machine learning methods. *IAES International Journal of Artificial Intelligence*, 13(1), 35–44.
9. Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4), 485–496.
10. Li, C., & Zheng, J. (2022). Software reliability prediction through encoder-decoder recurrent neural networks. *International Journal of Mathematical, Engineering and Management Sciences*, 7(3), 325–340.
11. Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27, 504–518.
12. Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), 2–13.
13. Qamar, T., & Bawany, N. Z. (2023). Understanding the black-box: towards interpretable and reliable deep learning models. *PeerJ Computer Science*. <https://doi.org/10.7717/peerj-cs.1629>
14. Rashid, J., Mahmood, T., & Nisar, M. W. (2019). A study on software metrics and its impact on software quality. *arXiv:1905.12922*.
15. Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2013). Data quality: Some comments on the NASA software defect datasets. *IEEE Transactions on Software Engineering*, 39(9), 1208–1215.

16. Siddika, A., Begum, M., Al Farid, F., Uddin, J., & Karim, H. A. (2025). Enhancing software defect prediction using ensemble techniques and diverse machine learning paradigms. Eng, 6(7), 161.
17. Zhang, Y., & Liu, N. (2025). Investigation and research on several key issues of software defect prediction. IET Software, 2025, Article 6615496.