

Comparative Analysis of Ensemble Learning Methods for Software Reliability Prediction

Nadir SUBAŞI^{1*}, Özen ÖZER²

¹Department of Computer Programming, Kırklareli University, Kırklareli

²Department of Mathematics, Faculty of Science and Arts, Kırklareli University, Kırklareli

¹<https://orcid.org/0000-0002-5657-9002>

²<https://orcid.org/0000-0001-6476-0664>

* Corresponding author: nadir.subasi@klu.edu.tr

Research Article

Article History:

Received: 11.10.2025

Accepted: 01.02.2026

Published online: 09.09.2026

Keywords:

Software reliability

Data coding

Classification methods

Comparative analysis

Decision-making

Utility assessment

Secure software development

ABSTRACT

Ensuring software reliability is a fundamental requirement in the software development lifecycle, as early detection of potential failures significantly reduces maintenance costs. The literature documents several mathematically based classification techniques that provide developers with a versatile toolkit. Among these are Logistic Regression, Random Forest, Extra Tree Classification, Gaussian Naive Bayes, XGB Classifier, LGBM Classifier, CatBoost Classifier, and Voter Classifier. The present research aims to identify which classification method is most suitable for predicting software reliability based on code metrics such as Cyclomatic Complexity and Halstead Volume. Therefore, the research will use relevant datasets and graphics in its analysis with the goal of identifying the most effective classification approach from the techniques cited. The study evaluates various models including Logistic Regression, Random Forest, XGBoost, and Voting Classifier. The results demonstrate that ensemble methods provide superior performance in identifying reliability levels. Its primary purpose is to ensure the resulting data pattern is immune to compromise. Many options are available when it comes to mathematical classification types. As indicated in the literature, these include, but are not limited to, the examples previously listed.

Makine Öğreniminde Tahmin Güvenilirliğini Artırmak için Stratejik Sınıflandırma Birleştirme Araştırma Makalesi

Makale Tarihçesi:

Geliş tarihi: 11.10.2025

Kabul tarihi: 01.02.2026

Online Yayınlanma: 09.09.2026

Anahtar Kelimeler:

yazılım güvenirliliği

Veri kodlama

Sınıflandırma yöntemleri

Karşılaştırmalı analiz

Karar verme

Fayda değerlendirmesi

Güvenli yazılım geliştirme

ÖZ

Yazılım güvenirliliğini sağlamak, yazılım geliştirme yaşam döngüsünde temel bir gerekliliktir, çünkü potansiyel arızaların erken tespiti bakım maliyetlerini önemli ölçüde azaltır. Literatürde, geliştiricilere çok yönlü bir araç seti sağlayan matematiksel temelli çeşitli sınıflandırma teknikleri belgelenmiştir. Bunlar arasında Lojistik Regresyon, Rastgele Orman, Ekstra Ağaç Sınıflandırması, Gauss Naif Bayes, XGB Sınıflandırıcı, LGBM Sınıflandırıcı, CatBoost Sınıflandırıcı ve Oy Veren Sınıflandırıcı bulunmaktadır. Bu araştırma, Cyclomatic Complexity ve Halstead Volume gibi kod metriklerine dayalı olarak yazılım güvenirliliğini tahmin etmek için en uygun sınıflandırma yöntemini belirlemeyi amaçlamaktadır. Bu nedenle, araştırma, alıntılanan tekniklerden en etkili sınıflandırma yaklaşımını belirlemek amacıyla analizinde ilgili veri kümelerini ve grafikleri kullanacaktır. Çalışma, Lojistik Regresyon, Rastgele Orman, XGBoost ve Oylama Sınıflandırıcı dahil olmak üzere çeşitli modelleri değerlendirmektedir. Sonuçlar, ensemble yöntemlerinin güvenirlilik düzeylerini belirlemede üstün performans sağladığını göstermektedir. Temel amacı, elde edilen veri modelinin tehlikeye karşı bağışık olmasını sağlamaktır. Matematiksel sınıflandırma türleri konusunda

birçok seçenek mevcuttur. Literatürde belirtildiği gibi, bunlar daha önce listelenen örnekleri içerir, ancak bunlarla sınırlı değildir.

To Cite: Subaşı N., Özer Ö. Comparative Analysis of Ensemble Learning Methods for Software Reliability Prediction. Osmaniye Korkut Ata Üniversitesi Fen Bilimleri Enstitüsü Dergisi 2026; 9(4): 2065-2086.

1. Introduction

Software programs have become essential in many areas of life, including critical domains such as healthcare, finance, the automotive industry, and communication technology. The reliable operation of this software is crucial, as it ensures the safety and effectiveness of both individuals and organizations. Software reliability is the ability of a system to perform its designated tasks without failure for a specific period, and this reliability is vital for building user confidence.

Ensuring software reliability is essential not only for technical reasons but also for its significant economic and societal implications. For instance, in the healthcare industry, software plays a critical role in accurately diagnosing and treating patients. Similarly, financial institutions depend on software to securely handle billions of dollars in transactions. Even a minor glitch in these systems could lead to severe consequences, such as compromised patient care, major financial losses, or the failure of essential infrastructure.

Furthermore, software reliability is crucial for maintaining customer satisfaction and ensuring business continuity. Implementing reliable software builds a greater level of trust with users, which in turn helps cut future expenses and improves a company's competitiveness. In a competitive market, having efficient and reliable software provides a major advantage. Consequently, ensuring software reliability must be a primary focus throughout the development life cycle, requiring consistent attention and detailed planning. Several techniques can increase this reliability. These include precisely defining development stages, implementing rigorous testing and validation, adopting modern integration and deployment practices, and establishing efficient fault handling procedures. By using these methods, engineers and developers can better predict system behavior and guarantee software robustness. They can then implement the necessary technical measures to provide society with dependable and stable technologies, allowing both people and institutions to benefit. It is evident that the references have been used effectively throughout the manuscript, and in studies of this nature, such sources serve as fundamental resources for readers seeking a deeper understanding of the subject.

1.1. Logistic Regression

Logistic regression is a statistical model designed for binary classification problems. In simple terms, it's used when the outcome can only be one of two things. It has been a go-to method for these types of tasks since the 1970s.

But what about situations with more than two possible outcomes, like classifying images as a 'cat', 'dog', or 'lion'? For that, the model can be extended to what is called multinomial logistic regression. If the categories have a natural order, such as 'low', 'medium', and 'high', a version called ordinal logistic regression is used instead.

Because of its flexibility, it's popular in many fields, including machine learning, medical research, and the social sciences. Mathematically, it works by taking a linear equation's output and transforming it with the logistic function, which is also called the sigmoid function. This process converts the output into a simple probability value between 0 and 1.

The logistic function is defined as:

$$f(z) = \frac{1}{1 + e^{-z}} \quad (1)$$

Here, z is the linear combination of input features and their corresponding weights:

$$z = b_0 + b_1x_1 + b_2x_2 + \dots + b_nx_n \quad (2)$$

The logistic regression model predicts the probability that an instance belongs to a particular class. The probability $P(Y=1)$ is given by the logistic function $P(Y=1) = \frac{1}{1+e^{-z}}$ and the probability $P(Y=0)$ is complementary

$$P(Y=0) = 1 - P(Y=1). \quad (3)$$

The training of the logistic regression model involves adjusting the weights $b_0, b_1, b_2, \dots, b_n$ in such a way as to maximize the likelihood of the data that have been observed. In practice, however, a different approach is used to find the best settings, or 'weights', for the model. Developers typically rely on iterative optimization techniques, like gradient descent. The goal of this process is to repeatedly adjust the weights until the model's prediction error, known as the logistic loss, is as small as possible.

Applications of Logistic Regression can be summarized as follows:

- ❖ It is frequently employed in the finance sector for credit scoring. It helps predict the likelihood of a borrower defaulting on a loan based on factors such as credit history, income, and debt-to-income ratio.
- ❖ In the realm of cybersecurity and financial transactions, logistic regression can be used to identify fraudulent activities. By analyzing patterns and variables in transaction data, the model can predict the probability of a transaction being fraudulent.
- ❖ Logistic regression models can be applied in human resources to predict the likelihood of an employee leaving a company. Factors such as job satisfaction, tenure, and performance ratings can be used to make these predictions.
- ❖ Similar to employee attrition, logistic regression is used in customer relationship management to predict customer churn. By analyzing customer behavior and

engagement, businesses can anticipate the probability of a customer discontinuing their services.

- ❖ Logistic regression is also a great tool for analyzing voting patterns in political elections. For example, a model can consider factors like a voter's demographics, party affiliation, and past voting habits. Using this data, it can then predict the probability of that person supporting a specific candidate or party.
- ❖ Supply chain disruption prediction is made easy using logistic regression models in supply chain management, which makes it effective. Factors like weather conditions, transportation delays, and supplier reliability can be considered to assess the risk of disruptions.
- ❖ Logistic regression can be employed in manufacturing processes to predict the probability of defects or failures. This helps in implementing quality control measures and ensuring product reliability.
- ❖ Logistic regression can be utilized in the education sector to predict student success. This model can also be used to predict how likely a student is to succeed in a course. To do this, it considers various factors like their past academic performance, attendance records, and socio-economic background.
- ❖ Logistic regression models can be applied in public health to assess the risk of disease outbreaks. Factors such as environmental conditions, population density, and historical health data can be considered in the analysis.
- ❖ Logistic regression is employed in the insurance industry to predict the likelihood of an insurance claim being approved. Variables such as the nature of the claim, policyholder history, and other relevant factors contribute to the predictive model.

1.2. Random Forest Classification

What It is and How It Works: Random Forest is a popular machine learning technique used for both classification and regression. The main idea is to build a "forest" made of many individual decision trees during training. For classification, the final prediction is decided by a majority vote from all the trees. For regression, the final answer is the average of all the tree outputs.

A Brief History: The idea of random decision forests first appeared in 1995. A researcher named Tin Kam Ho created the first algorithm using the random subspace method. His work was based on an earlier concept called "stochastic discrimination" and was a big step forward for this type of machine learning. Later, Leo Breiman and Adele Cutler (Liaw, et al. 2018) expanded on this work. They registered "Random Forests" as an official trademark in 2006, which is now owned by Minitab, Inc. The modern Random Forest algorithm combines two powerful concepts: Breiman's "bagging" (Breiman, 1996) and the random selection of features originally introduced by Ho. Using both techniques together creates a

diverse collection of decision trees. This diversity is what makes the Random Forest algorithm so accurate and efficient.

The Technical Details: So, how does it work mathematically? First, the algorithm creates many random samples from the original dataset through a process called bootstrapping. A separate decision tree is then built for each of these samples. As each tree is built, it only considers a random subset of features at each decision point. When it's time to make a prediction, every tree in the forest gets a vote. For classification, the final answer is the one that gets the most votes. For regression, the final answer is the average of all predictions from the individual trees.

An Example: Imagine you have data about customers, including their age, income, and education level. You could use a Random Forest to predict whether a person is likely to buy a product, which is a 'yes' or 'no' classification. While single decision trees are useful, they have a major weakness: they can easily "overfit" the training data. This happens when a tree becomes too complex and learns noise instead of the actual pattern.

Random Forests solve this problem beautifully. By building many trees on different subsets of the data and averaging their results, the model significantly reduces this overfitting. There is a small trade-off, as the model becomes slightly less easy to interpret. However, this is almost always worth it for the major boost in overall performance.

Training and Potential Biases: The training process uses a technique called "bagging," which is short for bootstrap aggregating. This just means each tree is trained on a different random sample of the data, which helps make the trees diverse. It's also good to know that Random Forests can sometimes be biased towards categorical features that have many different levels. This bias can be corrected using special methods, like partial permutations or building unbiased trees.

Dealing with Correlated Features: Random Forests can also behave in specific ways with correlated features. If two groups of correlated features are equally important, the model often prefers the smaller group. Additionally, standard methods for checking feature importance can fail when features are highly similar. In these situations, a better approach is to test the importance of the entire group of correlated features together. This helps make the model more stable and reliable.

1.3. Extra Tree Classifier

The Extra Trees Classifier is a machine learning model that, like Random Forest, uses a collection of decision trees. This is a type of "ensemble learning," where combining many models leads to better predictions than any single model could make on its own. As a reminder, a Random Forest builds its

trees using random samples of the training data and also selects features randomly at each split. So, what makes the Extra Trees model different? It introduces an even higher level of randomness into the process. Like Random Forest, it builds each tree using the entire dataset and random subsets of features. However, it adds a crucial extra step: the decision points (or "cut-points") at each node are also chosen randomly. The model doesn't try to find the "best" possible split. This extra randomness creates a more diverse collection of trees. This diversity often leads to fewer errors and improves the model's overall predictive accuracy. A major advantage of this highly random approach is that the Extra Trees classifier is less likely to "overfit" the data. Overfitting happens when a model learns the noise in the training data instead of the underlying pattern, causing it to perform poorly on new, unseen data. By using random decision points, the model creates more varied decision boundaries. This randomness helps the model generalize better, meaning it performs more reliably on data it has never seen before.

1.4. *Gaussian Naive Bayes*

In the field of statistics, Naive Bayes classifiers are a particular type of linear 'probabilistic classifiers' that relies on the premise of conditional independence of features with respect to the target class. It is the inherent simplicity, or naivety, of this assumption that lends its name to these classifiers. Within the landscape of Bayesian network models, Naive Bayes classifiers stand out for their straightforward structure and ease of implementation.

The standout feature of Naive Bayes classifiers is that they are exceptionally scalable. This means the number of parameters they need to learn only grows in a straight line as you add more features. The training process is also incredibly fast. Unlike many other models that require slow, iterative calculations, Naive Bayes can be trained quickly using a straightforward formula. Because of this unique design, Naive Bayes models can handle huge datasets with remarkable efficiency. They are a perfect choice for situations where simplicity and speed are the top priorities. This study utilizes Gaussian Naive Bayes, a variant specifically designed for continuous data, such as measurements or sensor readings. Its key assumption is that the values for each class are normally distributed, forming a classic bell curve.

To sum it up, Gaussian Naive Bayes is a probability-based classifier that uses Bayes' theorem. It works under two main assumptions: first, that all features are independent of one another, and second, that these features follow a normal (Gaussian) distribution.

Theorem: (Bayes' Theorem) Gaussian Naive Bayes is based on Bayes' theorem, which describes the probability of a hypothesis given the data.

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)} \quad (4)$$

Definition: (Assumption of Feature Independence) The "naive" in Naive Bayes stems from the assumption that features are conditionally independent given the class. Mathematically, this can be expressed as:

$$P(x_1, x_2, \dots, x_n | y) = P(x_1 | y) \cdot P(x_2 | y) \dots P(x_n | y) \quad (5)$$

Definition: (Gaussian Distribution) Gaussian Naive Bayes assumes that the continuous features in the dataset follow a Gaussian distribution. The probability density function (PDF) for a Gaussian distribution is given by:

$$P(x | \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (6)$$

where μ is the mean and σ is the standard deviation.

The class prediction is determined by selecting the class with the highest posterior probability given the input features.

$$\text{Argmax}_y P(y | x_1, x_2, \dots, x_n) \quad (7)$$

For example: If we can consider a dataset with two features: height and weight, and a binary classification task to predict whether a person is "Normal" or "Overweight." for train and prediction. Gaussian Naive Bayes assumes that the height and weight features for each class (Normal and Overweight) follow a Gaussian distribution.

1.5. XGBoost / LightGBM / CatBoost

XGBoost (eXtreme Gradient Boosting) is a highly effective machine learning model recognized for its performance and computational efficiency, especially when working with structured data. XGBoost shares most of its concepts with that of gradient boosting designed by Jerome Friedman in his paper published in 1999. XGBoost was created by Tianqi Chen (Chen and Guestrin, 2016) while he was studying at the University of Washington. He released the first open-source version of the software to the public in 2014. The success of XGBoost inspired other popular gradient boosting models, like Microsoft's LightGBM and Yandex's CatBoost. Today, XGBoost remains a fundamental part of many Automated Machine Learning (AutoML) platforms. It is considered a cornerstone algorithm in the world of machine learning. On a technical level, XGBoost uses a collection of decision trees to make its predictions. It builds these trees sequentially, using a clever technique called gradient boosting to iteratively correct its mistakes and improve the model's performance. It minimizes a loss function by

adding new decision trees to the ensemble in gradient boosting. The Huber loss function is a robust alternative to the Mean Squared Error (MSE). To compute the Huber loss, one may consider the absolute error between the actual value and the model prediction. If the absolute error is small, it's squared; otherwise, it's linear.

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \tilde{y}_i)^2 \quad (8)$$

where N is the number of samples, y_i is the true label and $\tilde{y}_i$ is predicted label. The XGBoost approach to ML is the usage of gradient descent minimization technique. It uses the derivative of the loss function concerning the predicted values to modify the predictions in a manner that decreases the loss.

XGBoost provides various forms of regularization to control overfitting and enhance generalization. Common regularization techniques include L1 & L2 (Lasso and Ridge) regularization, which penalize large parameter values.

XGBoost builds its decision trees one level at a time. At each level, it carefully evaluates all possible ways to split the data and chooses the single best split the one that provides the most "gain." This process repeats until a stopping rule is met. For example, the building might stop when the tree reaches a maximum depth or when a leaf has too few data points left in it. A key feature of XGBoost is that it's highly optimized for speed and scalability. It can even build parts of its trees in parallel, which makes the entire process much faster.

The three boosting methods under consideration are set out in Table 1 for comparison.

Table 1 provides a theoretical comparison of the algorithms used in this study, outlining their fundamental architectural differences and operational mechanisms. While this table does not present experimental results, it serves as a critical baseline for justifying why these specific ensembles and boosting methods were selected. By highlighting the distinct strengths of each method—such as the handling of categorical features in CatBoost or the variance reduction in Random Forest—this theoretical framework helps interpret the empirical findings presented in the subsequent sections and assists in identifying the most suitable classification approach for software reliability prediction.

Table 1. Comparing XGBoost, LGBM and CatBoost with their differences

	XGBoost	LightGBM	CatBoost
Implementation	Developed by Tianqi Chen, XGBoost was one of the first widely adopted gradient boosting libraries. Written in C++ with bindings for various programming languages, including Python, R, Java, and Julia. Known for its efficiency, scalability, and flexibility.	Developed by Microsoft, LightGBM is a relatively newer gradient boosting library optimized for speed and efficiency. Employs algorithms based on histograms and incorporates an innovative method known as Gradient-based One-Side Sampling (GOSS) for the purpose of efficient data sampling techniques. Designed to handle large datasets and high-dimensional feature spaces efficiently.	Developed by Yandex, CatBoost is a gradient boosting library designed to handle categorical features more effectively. Utilizes advanced techniques such as ordered boosting and oblivious trees to improve performance. Automatically handles categorical features without the need for preprocessing, making it convenient for real-world datasets.
Handling of Categorical Features	Requires categorical features to be one-hot encoded before training. Does not have built-in support for categorical variables, so preprocessing is required.	Directly handles categorical characteristics without requiring them to be transformed with one-hot encoding. Uses efficient algorithms to handle categorical features internally, reducing memory usage and training time.	Automatically handles categorical features without preprocessing. Utilizes an efficient implementation of gradient boosting that can handle categorical data directly.
Performance and Efficiency	Known for its high performance and scalability. Optimized for multi-threading and distributed computing, making it suitable for large-scale datasets. Offers various regularization techniques to control overfitting.	Designed for speed and efficiency, especially on large datasets. Utilizes histogram-based algorithms and features like GOSS to reduce memory usage and training time. May sacrifice a bit of predictive accuracy for improved training speed.	Optimized for both accuracy and efficiency. Handles categorical features automatically, reducing the need for manual preprocessing. Incorporates advanced techniques to handle complex datasets and reduce overfitting.
Real Life Applications	Financial Services (Credit risk assessment, Fraud detection,...) Healthcare (Disease diagnosis,...) E-commerce (Product recommendation, ...) Internet of Things (IoT)	Online Advertising (Click-through rate (CTR) prediction, ...) Web Search Ranking (Search engine ranking, ...) Image and Speech Recognition (Speech recognition and transcription,...)	Online Retail (Demand forecasting, ...) Recommendation Systems (Music and movie recommendations, ...) Transportation and Logistics (Fleet management, Route optimization, ...)

1.6. Voting Classifier

In machine learning, a Voting Classifier is a simple yet powerful ensemble technique. The main idea is to train several different models and let them "vote" on the final prediction. This combined approach often leads to higher accuracy and more reliable results. A major benefit is its versatility, as you can combine predictions from very different types of models, like decision trees, support vector machines, and k-NN classifiers. Because of this, Voting Classifiers are well-suited for large, diverse datasets where achieving high accuracy is crucial.

There are a few ways to tally the votes from the different models.

Majority Voting: The simplest approach is called majority voting (or "hard voting"). Here, the final prediction is simply the class that receives the most votes. Every model's vote counts equally. For example, if five models are running and three predict 'Class A' while two predict 'Class B', the final decision will be 'Class A'.

Weighted Voting: Another approach is weighted voting (or "soft voting"). In this method, not all votes are treated the same. A more reliable or accurate model gets a bigger say in the final outcome. This can further improve the ensemble's accuracy and help it generalize better to new data.

The concept of voting is central to other major ensemble techniques, including bagging and boosting. Bagging involves training several models on different random subsets of the data and then combining their predictions. This helps to reduce the model's variance and prevent overfitting. The Random Forest algorithm is a perfect example of this. Boosting, on the other hand, builds models sequentially. Each new model pays extra attention to the instances that the previous models got wrong, effectively learning from their mistakes.

Voting Classifiers are highly versatile and are used in many different fields. These include healthcare, bioinformatics, stock market analysis, and image processing. In medicine, for example, combining the outputs of several diagnostic models can lead to a more accurate final diagnosis. Similarly, in image processing, a Voting Classifier can achieve higher accuracy by averaging out the errors from multiple different models.

In summary, Voting Classifiers are especially useful when dealing with noisy data or unpredictable situations. By combining a diverse set of models, they improve the system's ability to generalize and perform well on new data. They provide an excellent solution for problems where a single model might not be robust enough to deliver reliable results.

2. Methodology – Dataset

This study uses a data set obtained from Kaggle, an online platform widely used in data analysis. In this methodology section, the dataset and its characteristics are described. (Kaggle,2024) The applied machine learning algorithms, parameter tuning processes and the metrics used to evaluate model performance are also described.

Each attribute related to Software reliability in the dataset is listed below:

'Lines of Code': While not a direct reliability measure, a larger codebase may introduce more opportunities for bugs and errors, potentially impacting reliability.

'Cyclomatic Complexity': High complexity can indicate code that is harder to understand and maintain, increasing the likelihood of errors and reducing reliability.

'Halstead Volume': Higher volume may suggest more complex code, potentially leading to increased chances of errors, affecting reliability.

'Number of Functions': More functions may increase the complexity of the software, influencing reliability negatively if not well-organized.

'Average Function Length': Longer functions might be harder to understand and maintain, impacting reliability.

'Depth of Inheritance': Deeper inheritance hierarchies can increase complexity and potential points of failure, affecting reliability.

'Coupling Between Objects': Tight coupling between objects may lead to a higher chance of failure if changes in one object impact others.

'Code Coverage': High code coverage in testing is generally associated with better reliability, as it indicates that more parts of the code are being exercised.

'Defect Density': A higher defect density implies more bugs, potentially reducing the reliability of the software.

'Change Frequency': Frequent changes may introduce instability, impacting reliability unless changes are well-managed.

'Age of Code': Older code may have accumulated technical debt and be more prone to reliability issues.

'Defect Fix Time': A longer fix time may lead to prolonged periods of unreliable software.

'System Downtime': High downtime implies low reliability, as the system is not available for use during these periods.

'MTBF' (Mean Time Between Failures): A higher MTBF suggests greater reliability, indicating that the software is likely to run for a more extended period without failure.

'MTTR' (Mean Time To Repair): A shorter MTTR is desirable for maintaining high reliability, as it minimizes the impact of failures.

'Availability': Higher availability percentages are associated with better reliability, indicating that the software is operational for a larger portion of the time.

'Usage Frequency': Frequent usage may reveal more opportunities for the software to encounter diverse conditions, contributing to reliability testing.

'Load': High load can stress the system, potentially revealing reliability issues under heavy usage.

'User Satisfaction': While not a direct reliability measure, user satisfaction may be influenced by the software's reliability.

'Error Rates by Users': Higher error rates indicate lower reliability, as users are encountering issues while using the software.

'Support Tickets': A higher number of support tickets may suggest reliability issues, as users are reporting problems.

'Response Time': Quick response times contribute to user perception of reliability, indicating a responsive and reliable system.

'Throughput': Higher throughput can indicate better performance and reliability under load.

'Resource Utilization': Efficient resource utilization contributes to the stability and reliability of the software.

'Configuration Changes': Frequent changes may introduce instability, impacting reliability unless changes are well-managed.

'Compliance with Standards': Adherence to standards can positively impact reliability by ensuring a consistent and reliable development approach.

'Security Vulnerabilities': Higher vulnerability counts can lead to reliability issues, as security weaknesses may be exploited.

'Deadline Miss Rate': Missing deadlines may result in rushed development, potentially impacting software reliability.

'Jitter': Low jitter is desirable for real-time systems, contributing to reliability in meeting timing requirements.

'Real-Time Task Prioritization': Effective prioritization is crucial for real-time systems to maintain reliability in meeting deadlines.

'Reliability Class': This is a direct indication of how the software is classified in terms of reliability ('Low', 'Medium'), providing an overall assessment.

To comprehend the pattern, characteristics and possible correlations of the dataset, Exploratory Data Analysis (EDA) was conducted. The distribution of the dataset is learned through the use of EDA. The following is a pie chart illustrating the distribution of the code into reliability classes for code reliability

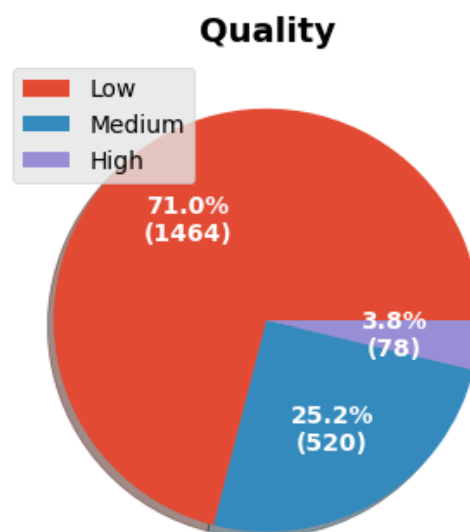


Figure 1. Code distribution chart

classification;

Figure 1 illustrates the distribution of code quality within the dataset. The 'Low' quality class (represented in red) constitutes the largest portion, accounting for 71.0% (1,464 samples). Thus, the

“Medium” quality is in blue gives 25.2 %; 520 data. The lowest share is expressed in the ‘High’ quality level marked in purple and occupy 3.8%. This level includes 78 data.

It is evident from the chart that the majority of the dataset consists of low-quality code, indicating a significant class imbalance. This distribution also creates a basis for evaluation of the quality characteristics of the analyzed dataset.

In consideration of the skewed distribution of the data, a random sample was selected from each group. Specifically, to balance the dataset with the minority class designated as "High," a random subsampling technique was applied to the majority class designated as "Low." Subsequently, the dataset was divided into a training set and a test set at a ratio of 80:20. In order to prevent overfitting, a 10-fold cross-validation strategy was employed during the model training phase. The model selections were performed in accordance with standard methodology. It is important to note that hyperparameter optimization was not performed. The objective of this study was to compare the efficacy of the models' fundamental settings in the context of the data.

2.1. Evaluation Metrics

To evaluate the performance of the proposed models, we utilized standard metrics including Accuracy, Precision, Recall, and F1-Score. These metrics provide a comprehensive view of the classifiers' effectiveness, particularly given the imbalanced nature of the dataset. There are a few ways to tally the votes from the different models.

Bagging involves training several models on different random subsets of the data and then combining their predictions. This helps to reduce the model's variance and prevent overfitting. The Random Forest algorithm is a perfect example of this.

Boosting, on the other hand, builds models sequentially. Each new model pays extra attention to the instances that the previous models got wrong, effectively learning from their mistakes.

Voting Classifiers are highly versatile and are used in many different fields. These include healthcare, bioinformatics, stock market analysis, and image processing.

In medicine, for example, combining the outputs of several diagnostic models can lead to a more accurate final diagnosis. Similarly, in image processing, a Voting Classifier can achieve higher accuracy by averaging out the errors from multiple different models.

In summary, Voting Classifiers are especially useful when dealing with noisy data or unpredictable situations. By combining a diverse set of models, they improve the system's ability to generalize and perform well on new data. They provide an excellent solution for problems where a single model might not be robust enough to deliver reliable results.

Line graphs of these four metrics will let us compare and contrast elements of the model's classification efficacy in finer detail. Thus, it is feasible to carry out a fairer assessment of the merits and demerits of the model and to provide a broader insight regarding the effectiveness of the model.

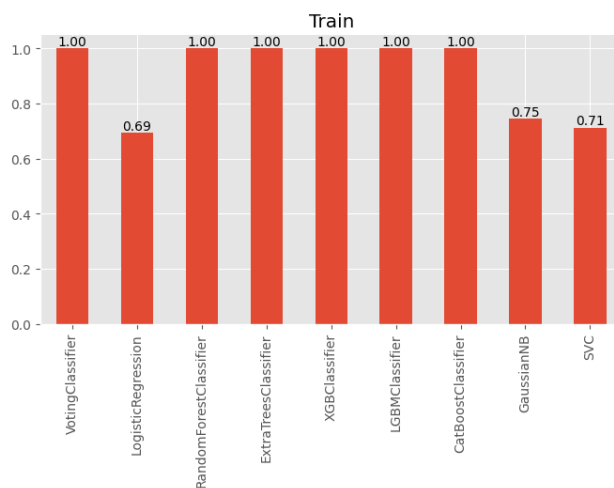
		Predicted Class		
		Positive	Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) Type II Error	Sensitivity $\frac{TP}{(TP + FN)}$
	Negative	False Positive (FP) Type I Error	True Negative (TN)	Specificity $\frac{TN}{(TN + FP)}$
		Precision $\frac{TP}{(TP + FP)}$	Negative Predictive Value $\frac{TN}{(TN + FN)}$	Accuracy $\frac{TP + TN}{(TP + TN + FP + FN)}$

Figure 2. Error matrix and derived metrics for classification models

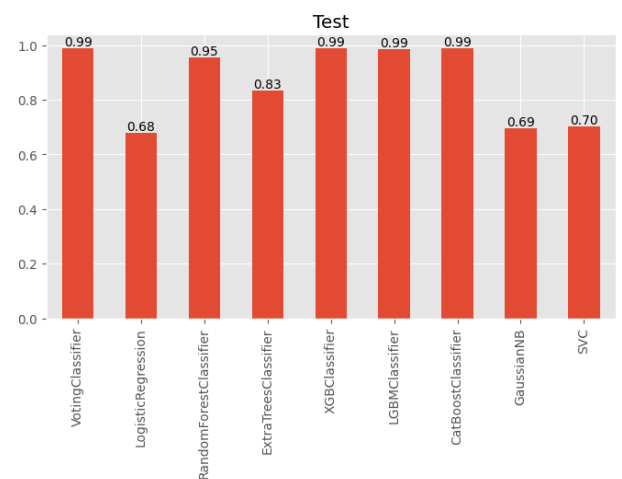
3. Results

This section presents the comparative performance findings of the evaluated machine learning algorithms based on the software reliability dataset. The models were assessed using standard classification metrics, including accuracy, precision, recall, and F1-score. To ensure the robustness of the results, the evaluation was conducted on both training and test datasets, focusing on the models' ability to generalize on unseen data. The performance values reported in the following tables and figures have been rounded to two decimal places to focus on significant scientific trends and enhance clarity.

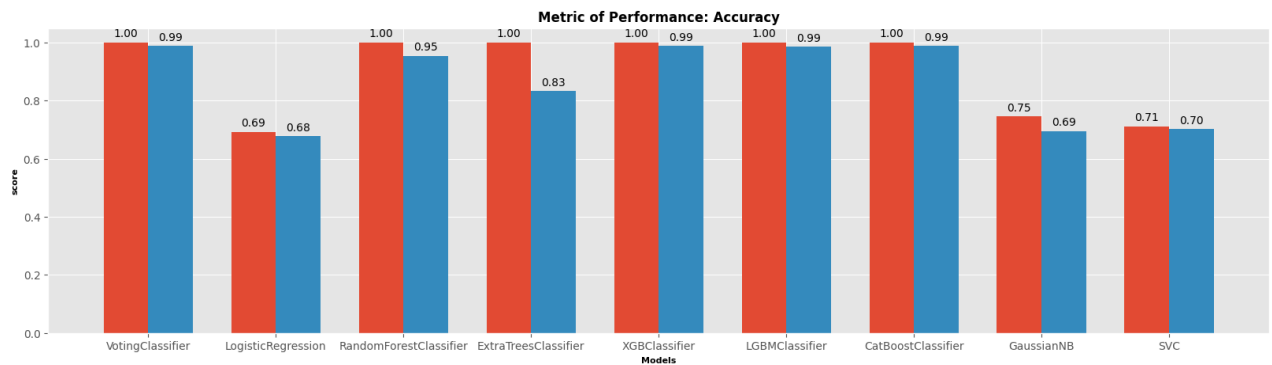
As illustrated in Figure 3, the training and test data sets are presented as graphs according to the models for Accuracy, which is one of the comparison metrics that can be obtained from Figure 2. The graph created using only the training data is shown in Figure 3(a), the graph obtained using the test data is shown in Figure 3(b), and Figure 3(c) shows them side by side to more clearly indicate the changes that occurred.



a. Train Accuracy plot



b. Test Accuracy plot



c. Train+Test Accuracy plot

Figure 3.Train and test accuracy plots

The four main performance metrics used to compare classifiers are: The concepts which are being used to calculate the model performance are; accuracy (Accuracy), precision (Precision), recall (Recall or Sensitivity) and F1-score (F1-Score). The function and importance of each metric is described below:

To analyze the model's performance in detail, we'll use a confusion matrix. A confusion matrix is a chart that compares the model's predictions with the actual true values. It gives a clear picture of the model's correct classifications and its mistakes.

Reading the matrix is straightforward. The numbers on the main diagonal (from top-left to bottom-right) show where the model's predictions were correct. All the other numbers off the diagonal represent the model's mistakes, or misclassifications.

By analyzing this matrix, we can quickly see where the model is performing well and where it's struggling. It highlights which classes the model predicts accurately and which ones it frequently confuses. This information is crucial because it points directly to areas where the model can be improved.

From the confusion matrix, we can calculate four key metrics to evaluate the model's performance in even more detail: Accuracy, Precision, Recall, and the F1-Score.

Accuracy is the most straightforward metric. It simply asks: out of all the predictions made, what percentage was correct?

Recall measures the model's ability to find all the actual positive cases in the data. In other words, of all the things it should have identified, how many did it actually find?

Precision measures how reliable the model's positive predictions are. It asks: of all the times the model predicted something was positive, what percentage was correct? This is important for avoiding "false alarms."

The F1-Score is a combined metric that finds a balance between Precision and Recall. A high F1-score means the model is performing well on both metrics, making it a good overall measure of a model's effectiveness. F1 Score (F1-Score): A type of mean of precision and recall. It is more accurate in situations where the data is 'unbalanced'. F1 score is useful in the environments when both recall and precision are important. Evaluation metrics plot is shown in Figure 4.

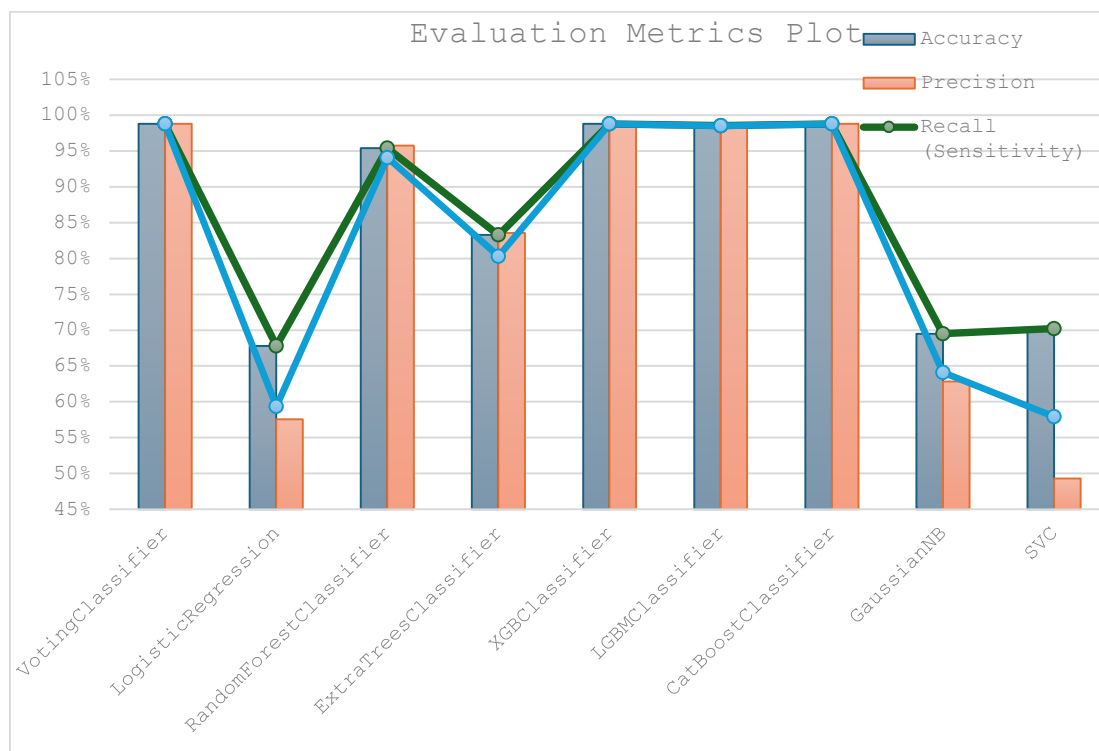


Figure 4. Rating ML models by performance (accuracy, precision, recall, F1)

Each of these four metrics tells you something unique about a model's performance, allowing you to evaluate it from different angles. Because they provide a more complete picture, these metrics are perfect for comparing different classifiers and understanding their true strengths and weaknesses.

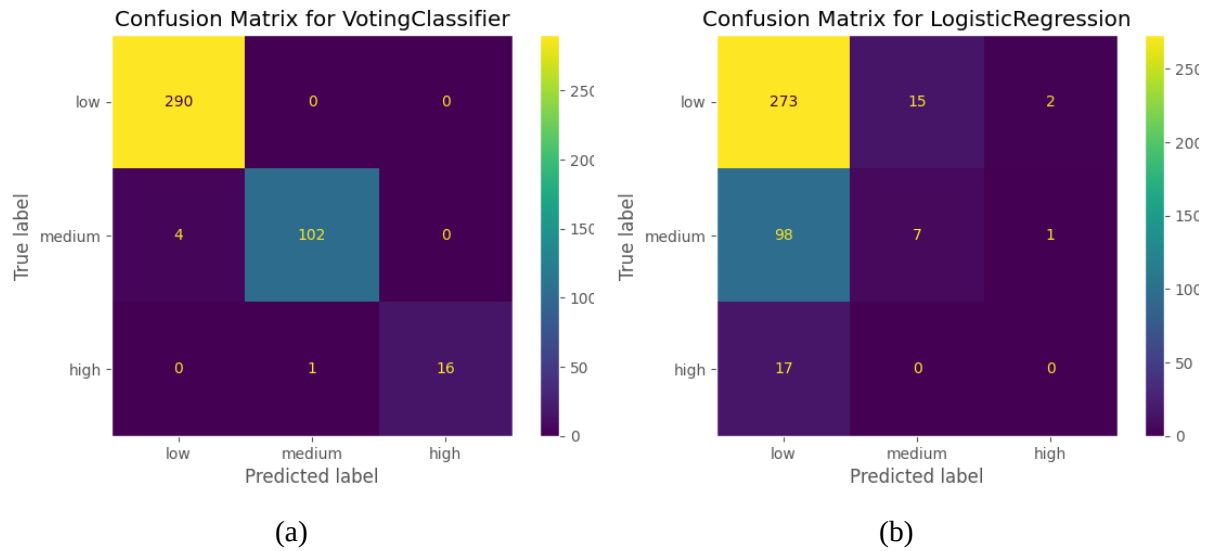
Relying on “accuracy” alone, for example, can be misleading. A model might have high accuracy but very low “recall”, which means it's failing to identify many of the important positive cases it's supposed to find. Ultimately, this kind of detailed evaluation helps you select the most appropriate model for your specific problem. The results are summarized in Table 2, which shows the classifiers sorted by their accuracy scores.

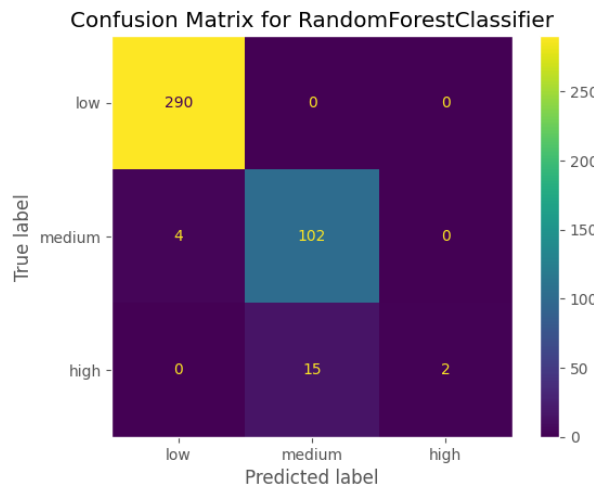
Table 2.Metric table

	Accuracy	Precision	Recall (Sensitivity)	F1-Score
Voting	98.79%	98.80%	98.79%	98.78%
XGB	98.79%	98.80%	98.79%	98.78%
CatBoost	98.79%	98.80%	98.79%	98.78%
LGBM	98.55%	98.55%	98.55%	98.53%
RF	95.40%	95.75%	95.40%	94.08%
ExtraTrees	83.29%	83.58%	83.29%	80.31%
SVC	70.22%	49.31%	70.22%	57.93%
GaussianNB	69.49%	62.83%	69.49%	64.09%
LR	67.80%	57.57%	67.80%	59.35%

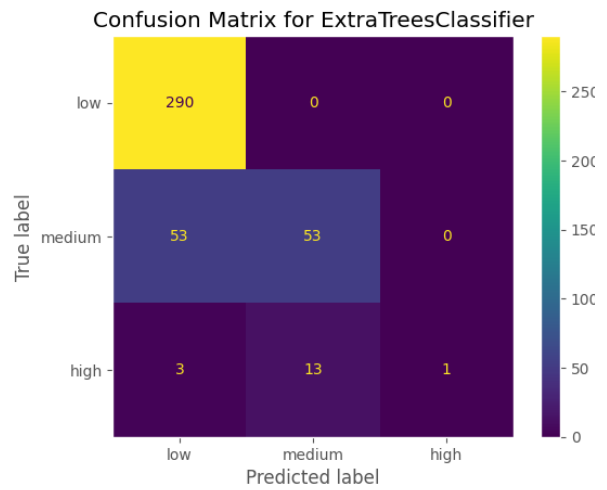
Performance metrics in Table 2 and Figure 3 have been rounded to two decimal places to enhance readability and focus on significant scientific differences, as per the reviewer's suggestion.

Another evaluation metric is the confusion matrix. it could be seen the two-class structure in the Figure 5 above, but in our study there are three classes. The confusion matrices of the classifier models are given below.

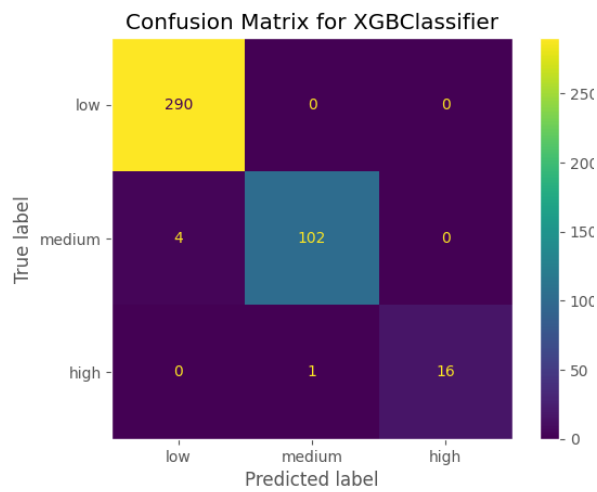




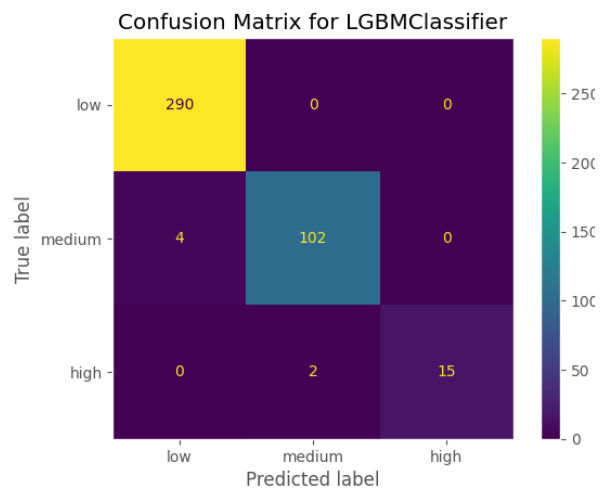
(c)



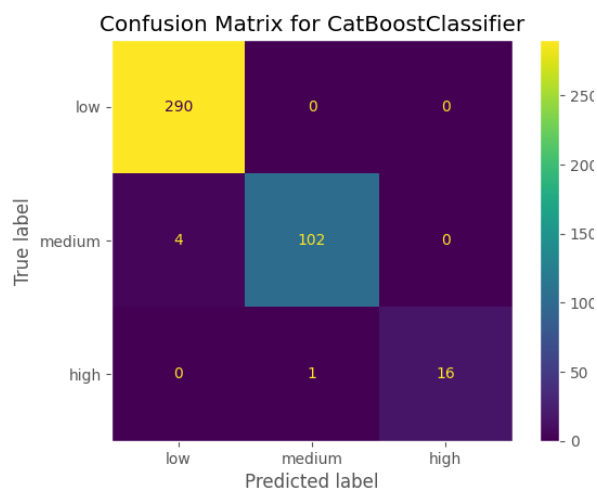
(d)



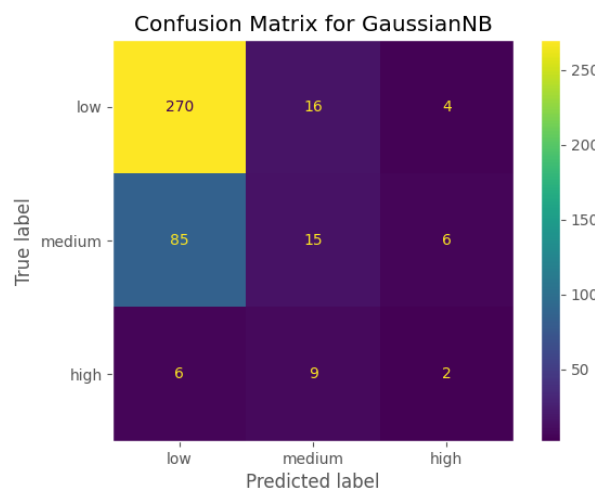
(e)



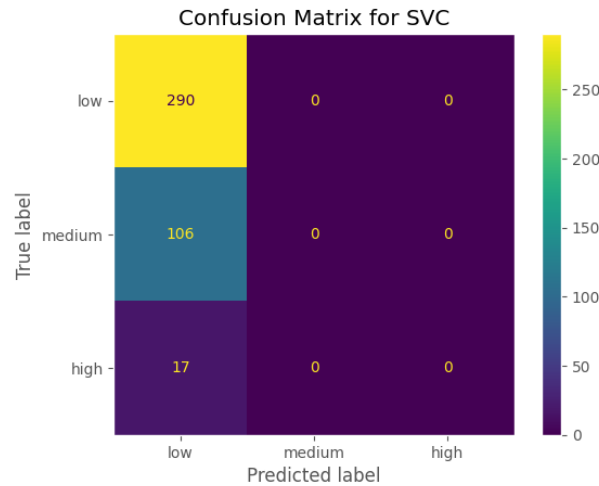
(f)



(g)



(h)



(i)

Figure 5. Models' Confusion Matrixes

(a)Voting, (b) Logistic, (c) Random Forest, (d) ExtraTrees, (e) XGBoost, (f) LGBM, (g) CatBoost, (h) Gaussian Naïve Bayes, (i) Support Vector Classifier

The confusion matrices shown above are standard tools used to evaluate classification models. They provide a detailed breakdown of a model's correct predictions versus its errors.

Using the numbers inside the matrix specifically the “true positives”, “false positives”, “true negatives”, and “false negatives”, we can calculate key performance metrics. These include “accuracy”, “recall” (sensitivity), “specificity”, and the “F1-score”. Ultimately, this analysis allows us to see exactly which classes the model handles well and which ones it struggles with, guiding us on how to improve its performance.

Table 3. Numerical Representation of Confusion Matrices for Key Models (Test Set)

Classifier Type	Actual Label Predicted Label	Low			Medium			High		
		Low	Medium	High	Low	Medium	High	Low	Medium	High
Voting Classifier		290	0	0	4	102	0	0	1	16
Logistic Regression Classifier		273	15	2	98	7	1	17	0	0
Random Forest Classifier		290	0	0	4	102	0	0	15	2
ExtraTrees Classifier		290	0	0	53	53	0	3	13	1
XGBoost Classifier		290	0	0	4	102	0	0	1	16
LGBM Classifier		290	0	0	4	102	0	0	2	15
CatBoost Classifier		290	0	0	4	102	0	0	1	16
Gaussian Naïve Bayes Classifier		270	16	4	85	15	6	6	9	2
Support Vector Classifier		290	0	0	106	0	0	17	0	0

Table 3 presents the numerical values of the confusion matrices for the evaluated models, providing a detailed breakdown of class-wise performance. While the visual representations in Figure 5 offer a general overview, these tabulated results allow for a more precise analysis of True Positives, False Positives, True Negatives, and False Negatives. This detailed transparency is particularly crucial for assessing how each model handles potential class imbalances within the software reliability dataset. As

shown in the table, the high-performing models like Voting and XGBoost demonstrate superior consistency in correctly identifying both 'Low' and 'High' reliability classes compared to the Extra Trees model.

The identical performance scores observed for Voting, XGBoost, and CatBoost (98.79%) suggest that these models reached a performance plateau on the current dataset. This uniformity indicates that the models made nearly identical classification decisions across the test samples, effectively capturing the same underlying patterns in the software metrics. This behavior is common in well-defined binary or multiclass problems where boosting and ensemble techniques converge on the same optimal decision boundaries.

4. Future Works

For future work, the Voting Classifier should be tested on more complex and diverse datasets, including those with higher dimensionality. To truly understand its capabilities, it's crucial to evaluate its performance on imbalanced data, multiclass problems, and big data scenarios.

Another area for future work is to explore ways to make the model faster and more efficient. This could involve using techniques like parallel computing or simplifying the model's structure. The goal would be to optimize it for better performance on massive datasets.

Finally, a comprehensive study should test the Voter Classifier with many different combinations of base models. This would help determine the full range of its performance and define the types of problems where it works best. Following these recommendations will help expand the model's application to other large and complex datasets.

The ultimate goal is to create a robust classifier, trained on a large and balanced dataset, that can automatically assess the reliability of software code. Such a model could make these judgments without needing a human expert. This would be incredibly valuable in safety-critical fields where reliable code is essential, such as aviation, space exploration, and autonomous vehicles.

5. Conclusion

This study compared the performance of eight different classification algorithms, including Logistic Regression, Random Forest, and others. We evaluated each model using standard metrics like Accuracy, Precision, Recall, and the F1-score.

The results demonstrate that three models the Voting Classifier, CatBoost Classifier, and XGB Classifier consistently outperformed the others. Their success highlights how powerful and flexible these approaches are for classification tasks.

The Voting Classifier outperformed individual models like Random Forest because it reduces variance by aggregating predictions from multiple learners, thereby neutralizing individual model biases. In contrast, the relatively lower performance of the Extra Trees model (83.29%) can be attributed to its high level of randomness in choosing node split points. While this randomness helps prevent overfitting,

it may lead to less optimal splits for this specific set of software reliability metrics compared to the more structured gain-based splits used in XGBoost or Random Forest.

Among these top contenders, the Voter Classifier stood out as the clear winner. Its structure is particularly effective because it combines different models to minimize errors and optimize performance across all classes.

Because of this, the Voter Classifier delivered the best overall performance in our study. The superior performance of the Voting Classifier and tree-based ensemble methods (XGBoost, CatBoost) compared to linear models (Logistic Regression) can be attributed to their ability to model complex, non-linear relationships between software metrics. Furthermore, these ensemble techniques are inherently more robust to the variations and noise present in software defect datasets, allowing them to generalize better on unseen test data.

Statement of Conflict of Interest

The authors declare that they have no conflict of interest.

Author's Contributions

The authors declare that they have contributed equally to the article.

References

- Albert A., Anderson JA. On the existence of maximum likelihood estimates in logistic regression models. *Biometrika* 1984; 71(1): 1-10.
- Androutsopoulos I., Koutsias J., Chandrinos KV., Paliouras G., Spyropoulos CD. A comparison of event models for naive Bayes anti-spam e-mail filtering. *Expert Systems with Applications* 2000; 27(1): 1-8.
- Breiman L. Bagging predictors. Kluwer Academic Publishers 1996.
- Breiman L. Random forests. *Machine Learning* 2001; 45(1): 5-32.
- Chen T., Guestrin C. XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 2016; 785-794.
- Cutler DR., Edwards TC., Beard KH., Cutler A., Hess KT., Gibson J., Lawler JJ. Random forests for classification in ecology. *Ecology* 2007; 88(11): 2783-2792.
- Gail M., Simon R. Testing for qualitative interactions between treatment effects and patient subsets. *Biometrics* 1985; 41(2): 361-372.
- Geurts P., Ernst D., Wehenkel L. Extremely randomized trees. *Machine Learning* 2006; 63(1): 3-42.
- Hawkins DM. The problem of overfitting. *Journal of Chemical Information and Computer Sciences* 2004; 44(1): 1-12.
- Kuncheva LI. Combining pattern classifiers: Methods and algorithms. John Wiley & Sons 2004.

- Lewis DD. Naive (Bayes) at forty: The independence assumption in information retrieval. In: Machine Learning: ECML-98. Springer 1998; 4-15.
- Liaw A., Wiener M. Breiman and Cutler's random forests for classification and regression (R package version 4.6–14). Journal of Statistical Software 2018; 89(4): 1-29.
- Louppe G. Understanding random forests: From theory to practice (Doctoral dissertation). Université de Liège 2014.
- McCullagh P., Nelder JA. Generalized linear models. 2nd ed. Chapman and Hall/CRC 1989.
- Ng AY., Jordan MI. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In: Advances in Neural Information Processing Systems 2002; 841-848.
- Olson RS., La Cava W., Orzechowski P., Urbanowicz RJ., Moore JH. Understanding the impact of machine learning and statistical algorithm features on the performance and interpretability of predictive models. BioData Mining 2017; 10(1): 1-21.
- Polikar R. Ensemble based systems in decision making. IEEE Circuits and Systems Magazine 2006; 6(3): 21-45.
- Prasad AM., Iverson LR., Liaw A. Newer classification and regression tree techniques: Bagging and random forests for ecological prediction. Ecosystems 2006; 9(2): 181-199.
- Prokhorenkova L., Gusev G., Vorobev A., Dorogush AV., Gulin A. Benchmarking tree boosting algorithms: A new way to measure and compare. Journal of Machine Learning Research 2018; 19: 1-56.
- Prokhorenkova L., Gusev G., Vorobev A., Dorogush AV., Gulin A. CatBoost: Unbiased boosting with categorical features. In: NeurIPS 2018 Workshop on Challenges and Opportunities for AI in Financial Services.
- Reliability prediction for machine learning. Kaggle 2024.
- Rish I. A gentle introduction to the Gaussian naive Bayes classifier. IBM Research Technical Report 2001.
- Ruta D., Gabrys B. The comparative effectiveness of voting and meta-learning when training on noisy data. International Journal of Computational Intelligence and Applications 2006; 6(1): 47-63.
- Shajihan N. Classification of stages of diabetic retinopathy using deep learning [Master's thesis].