Journal of Harbin Engineering University
ISSN: 1006-7043

Vol 46 No. 11
November 2025

Software Testing based on Random Forest and Adaboost Model

M. A. Abejide[1,2], A. O. Olagunju[2], G. B. Iwasokun[2,3], O. S. Aderibigbe[1], B. Aremo[2], T. E. Ogunbiyi[2], T. E. Ayowole[2], M. N. Ubaka[2], A. D. Abogunrin[2]

[1] Department of Computer Sciences, Lagos State University of Science and Technology, Lagos, Nigeria.
[2] Department of Computer Science and Information Technology, Bells University of Technology, Ota, Nigeria
[3] Department of Software Engineering, Federal University of Technology, Akure, Nigeria

Abstract

In this paper, Adaboost is integrated with Random Forest to form a new machine learning model, known as the Adaboost-Random Forest model.

Aims and Objective: To use Random Forest as a base model, and Adaboost as a meta-classifier for an adjustable and adaptable software testing process.

Methodology: The integrated model utilized 350 datasets from Kaggle.com, which were randomly split into two groups in an 80:20 ratio. The model was implemented using the Python programming language, with the training and testing of the models performed several times to achieve accurate results.

Result: Accuracy of 80%, a precision of 0.88, a recall of 0.68, and an F1-score of 0.71 were recorded for Random Forest. The integrated model (Adaboost-Random Forest) also had 85.5% accuracy, 0.86 precision, 0.81 recall, and 0.87 F1-score.

Conclusion: These results justified that the integrated Adaboost-Random Forest model is a more suitable model for software testing with its higher accuracy, recall, and F1-score, while the Random Forest only had higher precision. The Adaboost-Random Forest model achieved an increased accuracy of 5.5% compared to Random Forest, which is significant and demonstrates its suitability for high-accuracy testing in software engineering.

Keywords: Software Testing, Random Forest, Adaboost, Hybridization, Artificial Intelligence, Machine Learning

1. Introduction

According to [1], software is presently an essential component of human life. It is often a product and vehicle for conveying a product [2]. However, good software must be designed and constructed with minimal or no errors. Software errors often emanate from requirements gathering, design, and construction flaws, which can have great consequences, such as financial loss, lack of timeliness, and even losses of various degrees. Hence, every software needs to be authenticated and verified to establish its reliability and adequacy for the users. Basically, testing has to do with ensuring that a software product does nothing unintended and ascertaining quality and reliability. Software testing is also used to determine if the software meets the needs and specifications of the users [3]. In spite of all the progress and innovations in software development practices and programming languages, software testing, verification, and validation still stand as very important tasks [4].

Previously, software testing was done manually [5]. Manual testing requires the physical inspection of the application against users' necessities, in a manner devoid of automated testing tools. Testers carry out the validation process through the creation of manual test cases and applying them on their own [6]. The process of performing manual testing was highlighted in [5]. The process includes perusing the software and studying the application under test, drafting test cases which entail all the requirements cited in the documentation, reviewing all baseline and test cases with the team leader and in conjunction with the client, executing the test cases on the Application Under Test (AUT), reporting errors, and executing the failing test case to verify the pass. The aforementioned processes are characterized by a lot of challenges and problems, which include difficulty in testing every feature of the application, resource and time consumption, difficulty in coping with changes, software failure, and less test coverage, which requires skilled manpower [7]. Therefore, manual testing demands massive energy.

However, towards achieving faster and unceasing transfer of quality software, manual testing is insufficient, and automated testing has been taken as a better alternative by major software development industries [8].

Automated testing tools such as Katalon Studio, Selenium, QTP, Jineter, and Testlink are recommended only for stable systems and are mostly used for regression testing, that is, re-running useful and non-useful tests to achieve scenarios where pre-developed and tested applications consistently function as expected [9]. The challenges posed by automated software testing tools are the difficulty of integrating with other tools and getting resources and manpower with technical skills to correctly plan and maintain a test automation basis. Requirement changes too often, hence using automated software testing tools requires using several testing tools to effectively carry out testing, which may be expensive [10]. Furthermore, the Agile method of software development does not usually give enough room for the requirements of the software to be decided and captured before coding commences because time of delivery and innovation are the driving forces.

Since the automated testing tools have the aforementioned problems, there is a need to develop a testing model that would be adaptive to changes in the requirements and functionality of the software. Such a model would be capable of carrying out software testing without being specifically programmed to function with some set of requirements, but would adjust as the requirements change. Such a testing model is trained to study new functionality, document it as the software requirements, and use it as the basis for testing the reliability of the software. In such scenarios, a shift towards Artificial Intelligence (AI) is a viable option. Machine learning in software testing has the potential to reduce time spent on testing and improve the accuracy and reliability of the test results. It can also reduce human errors, automate repetitive manual tasks, and provide early and frequent bug and error detection [10]. Based on the aforementioned, this study focused on the implementation of the Random Forest and Adaboost models to test new software applications whose functionalities and requirements are not known in advance before the construction commences. Other objectives are to design a software testing model using Random Forest and Adaboost, implement the models using the Python Programming Language, and evaluate and compare the models. The study is significant because it led to improved software testing accuracy and reliability, and the fostering of quality software. Section two deals with a review of related works, while section three presents the research methodology. The implementation, results, and discussion are presented in Section 4. Section five focuses on the conclusion drawn and the recommendations.

2. Literature Review

In recent years, different methods and approaches have been used to test the functionalities and requirements of software products. The investigation on the application of the manual method of software testing carried out in [5] revealed that it would be very difficult to achieve higher accuracy in testing software products with manual testing, as it would be very hard to test all the functionalities and requirements. This agrees with the findings in [6, 11], who submitted that when both manual and automated software testing are implemented concurrently, accurate and reliable results are achievable. They also reported that limited resources and poor management could hamper successful test automation.

A study on migration from manual to automated testing carried out in [12] established that a huge number of resources are required for developing and maintaining any automated software testing tool. It was established that the manual method cannot guarantee high software testing accuracy, as well as the high cost often associated with automated testing, and posited that some tests could be executed manually if there is no viable alternative or automated when manual execution is impossible. In [13], the investigation on automated software tools and frameworks established that a fruitful and competent software testing project requires the use of a proper framework and methodology. However, selecting the appropriate tools that satisfy these key components for optimal software testing is very difficult to achieve because it requires significant effort, time, and strategy. This view was corroborated in [8] with the submission that selecting the right automation testing tool and the appropriate framework remains one of the most significant issues for automation testing.

A study on software testing with the use of different machine learning techniques based on the selection of 48 primary studies, which were divided into groups for

systematic mapping analysis, was conducted in [14]. The study revealed that machine learning algorithms have higher testing accuracy when compared to automated testing tools. However, since machine learning algorithms have their respective areas of strength and weakness, careful study is often required to know their suitability before implementation. Automated software testing using meta-heuristic techniques based on improved Ant algorithms was performed in [10]. Seven algorithms were tested and evaluated, and the result showed that the Genetic Algorithm (GA) is very effective in promoting coverage of test cases with a performance similar to the improved genetic algorithm Ant colony optimization, which has a good astringency. The failure of GA with random testing scenarios and real-time testing, as well as the effectiveness of state transition testing (STT), which is frequently employed in web-based software systems and ant algorithms, were also stated. A genetic algorithm-based assessment for test case prioritization was conducted in [15]. The average percentage of statement coverage (APSC) measure was used to display an algorithm's efficiency, while the test case was prioritised based on the best findings. From the assessment, it was revealed that the genetic algorithm offered effective and efficient test case prioritization. This position was also held in [10], where a genetic algorithm was used to automate software testing. In [16], research using the genetic algorithm on value-based test case prioritisation for regression testing was carried out. Test Case Prioritization (TCP) was evaluated using test case reduction, test case selection, and test case prioritization. The results of the findings revealed that TCP techniques outperform other methods for regression testing, which is the same view expressed in [10, 15].

In [17], a study on Side Blotched Lizard optimized Adaboost (SBLA), which is an Adaboost convolutional neural network on test case minimization and ordering for regression testing, was carried out. The study identified high-rate resource consumption and time as the problem of regression testing. An Ada-boost Convolutional Neural network (SBLA–Ada–boost CNN) model optimized for Side Blotched Lizards was deployed for solving the problem based on the defect 4J dataset. A study on optimal test suite selection in regression testing with test case prioritization using a modified Ant and Whale optimization algorithm was carried out in [18]. The study used prioritization techniques for information gathering regarding the test case's prior execution to acquire test case arrangements. Test cases were arranged in a way that maximizes their efficacy in achieving certain performance objectives using the test case ordering technique. The regression testing ensures that software updates, whether they be for bug fixes or new feature additions, don't negatively impact previously functional functionalities. Kernel fuzzy e-mean clustering was used to help cluster the created test cases and pertinent test cases, given priorities. Test case prioritization was used to figure out which sequence of test cases would increase the likelihood of finding source code errors using artificial neural network classification algorithms with test case ordering modifications. The whale optimization technique was employed for the weight optimization process.

A study using Random Forest classifiers, namely Random Forest rotation, to predict software faults was carried out in [19]. A multidimensional and scalable pre-processing method was used to reduce the complexity of the software feature's dimension, with several experiments performed using numerical and normal data, one after the other. Results of the experiments revealed that numerical data performs better than normal data and established that Random Forest rotation is not suitable for implementation with large amounts of data. A Random Forest algorithm was used in [20] to minimize test cases for object-oriented testing using WEKA, an open-source program to generate a mathematical model that was used in the process. The method depends on selecting the significant object-oriented metrics. The mathematical model identifies highly efficient and less efficient test cases when several experiments are performed. The results of the tryouts revealed the effectiveness of the model as it decreased the test cases that were less efficient by covering classes that were prone to errors. Moreover, it decreased the cost and time involved in test suite reduction. In [21], a study on software fault prediction using Random Forest was conducted. The K-fold technique was used to split the data into training and testing categories, and any optimal characteristic associated with a software defect was chosen using the F-score and a new set of data was created every time categories were randomly modified. The implementation showed some promising results, in addition to how increasing the number of trees in the forest can cause overfitting and poor results.

3. Research Methodology

This section focuses on the methods used in the implementation of a software testing model that is anchored on Random Forest and Adaboost. The model involves five major steps, namely data collection, normalization of data, information gain ratio, prediction, and evaluation.

3.1 Data Collection

The dataset used to test the efficiency of the Random Forest and Adaboost algorithms-based model was obtained from www.kaggle.com. The dataset is available for free download at (license-free) https://www.kaggle.com/datasets/sapal6/the-testcase-dataset/code?datasetId=982874&searchQuery=datset+for+teting+the+reliabiliy+of+a+softwae

Table 1. Test case attributes and description

Attributes	Description
Test Case	Software Requirement
R_Priority	Requirement Priority of a particular business requirement
FP	The function point of each testing task, which in our case are test cases against each requirement, covers a particular FP
Complexity	Complexity of a particular function point or related modules (the description of assigning complexity is listed below in this section)
Time	Estimated max time assigned to each Function Point of a particular testing task by QA team lead or sr.
Cost	Calculated cost for each function point using complexity and time with function point estimation technique to calculate cost based on the formula listed below: cost = "Cost = (Complexity * Time) * mean amount set per task or Function Point

It was pre-labelled according to the Kaggle repository and collected in line with the Kaggle data collection adjudication procedure. The dataset comprises 350 rows and 5 columns, and its attributes describe the test case characteristics, while its description presents the different software requirements features as shown in Table 1.

3.2 Hybridization of Random Forest with Adaboost model.

At the pre-processing stage, the data was randomly selected, cleansed, and normalized before analysis. Data cleaning was used to detect and correct inaccurate data, while data normalization was used to organize the set data to increase cohesion. The attribute verifier was used to check the feature of the dataset, while the set data was divided, and the ranker selected a subset of the relevant dataset for use. This is done by selecting the high or more relevant subset of data from the original set to enhance the model prediction or classification. When this process is completed, the set of data with a lower rank is eliminated. At the processing state, the higher set of data is further divided into training and test data, as the data imbalance verifier classified the dataset with a skewed class proportion. Classes that make up a large proportion are called majority classes, which form 80% of the dataset, while those that make up a smaller proportion are called minority which form 20% of the dataset. The synthetic minority oversampling technique (SMOTE) balances the data by increasing the number of cases in the dataset. The model (Random Forest and Adaboost) was trained to work together as a single model by complementing and augmenting for optimal predictions and classification. The output of the result is represented with a high or low procedure of the test case, while the test case with high precedence is represented as high, and the test case with low precedence is represented as low. For classification, the output of a random forest is obtained using Equation (1):

$$\hat{y} = |(T_1(x), T_2(x), T_2(x), \ldots\ldots, T_n(x))| \quad (1)$$

$\hat{y}$ is the predicted class and $T_i(x)$, represents the prediction of the ith tree for input x and $|\ \ |$ is a modulus operation. For regression, the output is derived using Equation (2):

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} T_i(x) \quad (2)$$

$\hat{p}$ is the predicted class and $T_i(x)$ is the prediction of the ith tree for input x. The AdaBoost operation commences with the passing of the same preliminary weights to all training samples and then, for each iteration, a puny classifier is trained on the data to arrive at more cogent samples with improved weights. The weights of misclassified occurrences are then

increased with a view to attaining greater influence on the subsequent iterations. On the contrary, the suitably classified weight occurrences are decreased, while the weak classifiers, determined by the error rate, are merged to form a weighted sum. The Adaboost algorithm works on an $X \times Y$ dataset as seen in Equation (3):

$$(X, Y) = \{((x_1, y_1), \ldots\ldots, (x_N, y_N))(x_i \in R^d, y_i \in \{-1, +1\}, i = 1, \ldots, N)\} \quad (3)$$

x_i is the feature vector of ith example, y_i is the class label, N is the number of features, R^d denotes the d-dimensional Euclidean interstellar, which is the set of all conceivable d-dimensional trajectories where each constituent is an actual number. The AdaBoost-Random Forest algorithm integrates the serial learning process of AdaBoost with the ensemble power of Random Forests. Random Forest is used as a weak learner within the AdaBoost framework. The integration commences with assigning equal weights, t_s to all the training samples. t_s is derived from the formula:

$$t_s = \frac{1}{N} \quad (4)$$

N is the total number of samples. For every iteration, $t = 1, 2, 3, \ldots, \rho$, where ρ represents the number of boosting iterations, a weak learner is trained based on the Random Forest model $G_t(y)$. The model is trained on the training data, where every tree within the Random Forest is constructed via the prevailing sample weights t_s. This leads to samples with bigger weights and higher chances of being added to the bootstrap samples used for training every tree in the Random Forest. The derivation of the error rate, r_e follows this operation, on the weighted training data based on the formula:

$$r_e = \sum_{s=1}^{N} t_s \cdot \lceil G_t\,(x_s)\; y_s \rceil \neq t_s \quad (5)$$

$\lceil(.)\rceil$ is the indicator function which evaluates to 1 if the condition is true, and 0 otherwise. t_s is the true label of the sample y_s. The learner coefficient $\beta_e$for the Random Forest, the value is calculated from:

$$\beta_e = \frac{1}{2}\log\left(\frac{1-r_e}{r_e}\right) \quad (6)$$

β_e reflects the significance of the current Random Forest in the overall integration with improved models having better values. Consequently, the weights of the training samples are updated for the succeeding iteration. Every inappropriately classified sample has its weight amplified, while the one that is appropriately classified has its weight lessened as follows:

$$w_{t+1}(s) = w_m(s).\exp\left(-\beta_t t_i G_t\,(x_s)\right) \quad (7)$$

Each of the resulting weights is then standardized to add to 1. The final integrated model $G(y)$ is a weighted average of all the trained Random Forests, and it is derived from:

$$G(y) = sign\left(\sum_{w=1}^{\rho} \beta_w\, G_w\,(y)\right) \quad (8)$$

The $sign$ function gives the overall class tag. The algorithm for the AdaBoost-Random Forest algorithm is presented below. The algorithm is based on the flowchart presented in Figure 1.

Start 1

Input: test case data of varying feature output.
REM: Expected prediction (test case of high precedence) is high, the test case with low precedence is low process
Step 1: Supply the data sets of test cases to the application
Step 2: Perform data collection
Step 3: Normalised the data
Step 4: Compute information gain ratio
Step 5: Apply 3 both Random Forest and Adaboost to make a prediction
Step 6: Display the prediction result
Step 7 stop

3.3 Model Evaluation

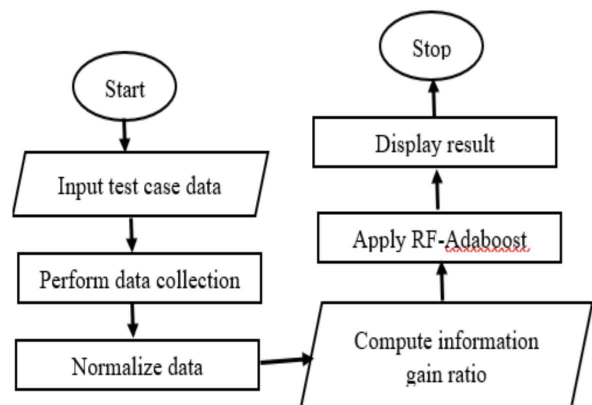

Figure 1. Flow chart for the Random Forest with Adaboost for software testing

One of the most important aspects of machine learning research is evaluation. When comparing the model's performance to other metrics, such as logarithmic loss, it may yield satisfactory results when assessed using an accuracy score. Having sourced the research dataset and test cases from the internet and successfully tested

the model, the *Accuracy, F1-score, Precision,* and *Recall* metrics were selected. Classification accuracy represents the percentage of correct guesses to the total number of input samples. F1 Score is used to estimate the test accuracy and strike an equilibrium between recall and precision, while Precision is the quotient of true positive results and the sum of positive predicted outcomes from the classifier, and the number of false positives. Recall gives the quotient of the accurate positive results and the sum of total positive and false negatives. The mathematical formulae for $Accuracy$, $F1\,Score$, $Precision$ and $Recall$ are presented below:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (9)$$

$$F1\,Score = \frac{2*TP}{2*TP+FP+FN} \quad (10)$$

$$Precision = \frac{TP}{TP+FP} \quad (11)$$

$$Recall = \frac{TP}{TP+FN} \quad (12)$$

TP, TN, FP, and FN represent the True Positive, True Negative, False Positive, and False Negative, respectively.

4. Implementation, Results, and Discussion

The experimental study of the model was carried out on a Core i7 HP Laptop with 16GB and 1TB HDD running on the Windows 10 Operating system. The Python programming language served as the frontend. The choice of the language is premised on its readability and ease of learning, extensive libraries and ecosystem, versatility, integrity, scalability, and strong community support. The dataset used for the study was obtained from Kaggle.com and contains 350 rows and 5 columns of data. It was split into training and testing sets in the ratio of 80% (280) and 20% (70). 5-fold cross-validation was adopted in order to split the data randomly into K-folds, and then the model was trained on the K-1 folds, while the remaining fold was used to test whether the predictor is effective or not. Five (5) random samples were taken, and the model was trained forty times. The training set in each of the partitions was used to train the model, while the test set in each of the partitions was used as hidden data for the model. In each training, one of the 5 random samples was selected for testing, and the results of each of the random samples were analyzed using accuracy, precision, recall, and F1-Score. These evaluation metrics were used because they were standard metrics used for this typical evaluation.

4.1 Results

Tables 2-5 show the subset of the actual status of each test case and the machine-predicted value for the repetitive test case data. If the machine prediction corresponds with the actual test case status with high precedence, a value T is returned; otherwise, it will return F. This was done for the entire forty random samples in the experiment. The model was trained forty (40) times, which implies forty experiments with their respective results. The subset results for four of the experiments are presented in Tables 3 to 6. In the first experiment shown in Table 2, the system predicted 61 correctly out of 70 test cases, and from Equation 1, the accuracy is $\frac{61}{70}\ x\ 100 = 87\%$. The system also gave a percentage rate of false negatives, which was calculated as 13%.

Table 2. The First test Case Experiment with actual Status and Predicted Result

S/N	Test Case	Actual Test case status	System prediction
1	Find out if listing in a directory is forbidden.	High	T
2	Check for assaults known as denial-of-service.	High	T
3	Check the application's logout functionality. Post logout, none of the post-login pages should be accessible.	High	T
4	It is recommended to transfer every credential via an encrypted route.	High	T
5	Special characters in the input should be ignored.	High	T
6	Applications and servers shouldn't be exposed to a page crash. This should bring up the error page.	Low	F
7	Examine memory and CPU utilization during periods of high load.	High	T
8	Determine the strain testing of the application. The system should not be disrupted under little strain. With high strains, the system should behave well.	Low	F

9	Verify how long it takes to execute a database query. The time limit for acceptable query execution shouldn't be exceeded.	Low	F
10	Examine how well triggers and stored procedures in the database are working.	Low	F
11	Determine to direct message functionality via TO, CC, and BCC pitches.	High	T
12	Check the duplicate name image upload.	High	T
13	Verify that there is a check box with the label remember password on the login page	Low	F
.	.	.	.
.	.	.	.
.	.	.	.
70	Determine the functionality of warehoused events and activities in the catalogue.	High	T

The subsets of the prediction results for the ninth, nineteenth, and twenty-eighth test case experiments are presented in Table 3, Table 4, and Table 5, respectively. 84%, 83% and 89% accuracies were recorded for the ninth, nineteenth, and twenty-eighth test cases, respectively.

Table 3. The Ninth Test Case Experiment with Actual Status and Predicted Result

S/N	Test Case	Actual Test case status	System prediction
1	Check by adding all the extensions or TLDs	Low	F
2	Determine spread functionality for archives with very large scopes.	High	T
3	Confirm through the addition of only the field name	Low	F
4	Confirm if the search domain is available and aligned	Low	F
5	Confirm the formatting for numeric or currency values.	High	T
6	The name of the transferred file must be in accordance with standards.	High	T
7	Confirm the correctness of the placeholder text’s spelling and grammar	High	T
8	Confirm if message functionality transferred to single, multiple, or distribution list receivers.	Low	F
9	Confirm the message headnote and footnote for the business logo, privacy policy, and other links.	High	T
10	Confirm the delivery of the message functionality based on the TO, CC, and BCC fields.	Low	F
11	Check by adding all the extensions or TLDs	High	T
12	Check that the search field is present and aligned	Low	F
13	Confirm that the record pitches are planned with the precise figures and attributes.	High	T
.	.	.	.
.	.	.	.
.	.	.	.
70	Messages with empty sender’s names are made impassable.	High	T

Table 4. The nineteen experiment

S/N	Test Case	Actual Test case status	System prediction
1	Confirm that the upload functionalities are working with the right file types, other than images, and that error messages are properly displayed.	Low	F
2	Confirm that the size of the uploaded image does not exceed the maximum.	Low	F
3	Messages with empty sender's names are made impassable.	High	T
4	Confirm that the name of the uploaded image has no spaces or any other invalid special characters.	High	T
5	Confirm any identical image names during upload.	High	T
6	Confirm that the size of the uploaded image does not exceed the maximum.	Low	F
7	Prior to committing data to the database, remove all the input fields before and after spaces.	High	T
8	Test stored procedures and triggers with sample input data.	Low	F
9	Determine if every one of the table constraints is correctly executed.	High	T
10	Verify if the database fields possess the right data type and length.	High	T
11	Verify if the message functionality has been delivered to a single, multiple or distribution list of receivers	Low	F
12	Verify that the search field is present and aligned	High	T
13	Verify that message functionality was delivered with TO, CC and BCC fields.	High	T
.	.	.	.
.	.	.	.
.	.	.	.
70	Verify the correctness of the radio button and drop-down list options in the database.	Low	F

Table 5. Twenty-eighth Experiment

S/N	Test Case	Actual Test case prediction	System prediction
1	Verify the login-related elements	Low	F
2	Verify the export functionality for documents with very large sizes.	High	T
3	Verify if the login page of the system is reactive.	High	T
4	There should be proper column names for the exported documents.	Low	F
5	Verify that numeric or currency values are properly formatted and in line with the one shown on the page.	High	T
6	The file name that conforms to the standard should be used for the exported Excel file.	Low	F
7	Confirm that there is a check box and a password label on the login page	High	T
8	Verify that the message is sent to single, multiple, or distribution list receivers.	High	T
9	Verify the existence of message head and footnotes for the company logo, privacy policy, and other links.	Low	F
10	Verify that messages are sent through the TO, CC, and BCC fields.	Low	F
11	Verify if the login page of the system is reactive.	Low	F
12	Verify that no duplicate images.	High	T

13	Verify the correctness of the formatting for numeric or currency values.	High	T
.	.	.	.
.	.	.	.
.	.	.	.
70	Ensure that no sender's name is empty.	High	T

Table 6 shows the results of precision, recall, F1, and accuracy for the forty experiments. While Table 7 presents the false negative results, Table 8 presents the performance evaluation of the Random Forest and Adaboost, and Random models.

Table 6. Random samples of training test cases using the hybridized Model

Number of Samples	Precision	Recall	F1	Accuracy %
1	0.87	0.82	0.91	87
2	0.84	0.80	0.86	84
3	0.86	0.81	0.89	86
4	0.83	0.79	0.81	83
5	0.87	0.82	0.91	87
6	0.86	0.81	0.89	86
7	0.84	0.80	0.86	83
8	0.83	0.78	0.81	83
9	0.84	0.81	0.86	84
10	0.86	0.81	0.89	86
11	0.87	0.82	0.91	87
12	0.86	0.81	0.89	86
13	0.84	0.80	0.86	84
14	0.86	0.81.	0.89	86
15	0.89	0.83	0.94.	89
16	0.86	0.81	0.89	86
17	0.84	0.80	0.86	84
18	0.86	0.81	0.89	86
19	0.83	0.78	0.81	83
20	0.87	0.80	0.88	87
21	0.85	0.81	0.86	85
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
39	0.84	0.81	0.86.	84
40	0.86	0.80	0.89	86
AVERAGE	**0.86**	**0.81**	**0.87**	**85.5**

Table 7. False negative using Random Forest and Adaboost

Random Samples	False Negative (%)
1	13.0
2	16.0
3	14.0
4	17.0
5	13.0
6	14.0
7	16.0
8	17.0
9	16.0
10	14.0
11	13.0
12	14.0
13	16.0
14	14.0
15	11.0
16	14.0
17	16.0
18	14.0
19	17.0
20	13.0
.	.
.	.
.	.
40	14.0
Average	**15.6**

Table 8. Performance Evaluation of Single Model (Random Forest and Adaboost) and Random Forest Model

Model	Avg. Accuracy %	Avg. Precision	Avg. Recall	Avg. F1 Score
Random Forest and Adaboost Model	85.5	0.86	0.81	87.0
Random Forest Model	80.0	0.88	0.68	71.0

4.2 Discussion

As shown in Figure 2, the Adaboost Model has an average accuracy of 85.5% for the dataset compared to the Random Forest model, which has an accura of 80%. This implies a significant improvement in the correctness of the proposed model compared with the correctness of the Random Forest. Figure 3 presents a precision value of 0.86 for the new model, while the

Random Forest model recorded a precision figure of 0.88. it could be inferred from these figures that the precision of the Random Forest model is greater than that of the proposed model, although the difference of 0.02 has no significant implication on the accuracy. Figure 4 presents the *Recall* of 0.86 and 0.68 for the new and Random Forest models, respectively. This translates to more accurate identification of positive instances by the new model. Figure 5 also shows that the proposed model has an F1 score of 87% while the Random Forest model has an F1 score of 71%, which establishes that the proposed model has a noteworthy improvement compared with the Random Forest model. Through the study, the risk of overfitting is degraded and reduced, which also leads to a positive effect on the accuracy of the software product.

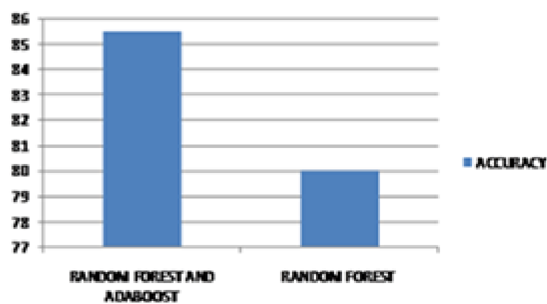

Figure 2. Accuracies performance evaluation of Adaboost-Random Forest model and Random Forest model

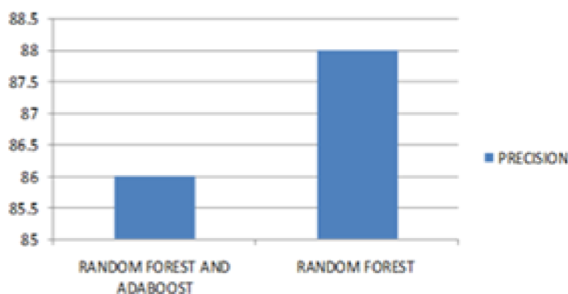

Figure 3. Precisions Performance Evaluation of Adaboost-Random Forest model and Random Forest model

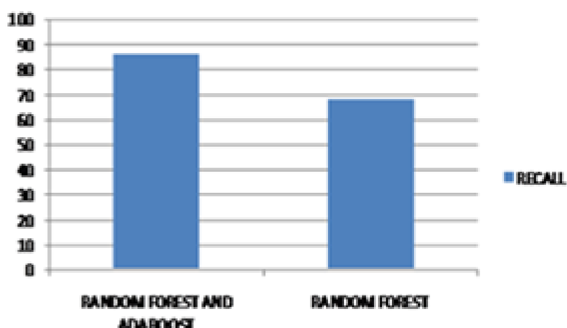

Figure 4. Recalls Performance Evaluation of Adaboost-Random Forest model and Random Forest model

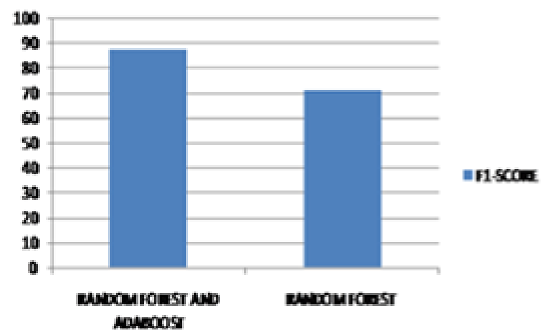

Figure 5. F1-score Performance Evaluation of Adaboost-Random Forest model and Random Forest model

In software engineering, testing often focuses on cost and risk reduction, fast access to the market, establishing compliance with requirements, quality and performance optimization, and meeting customers' satisfaction, all of which require the formulation of flawless and error-free code. To achieve these, very highly accurate platforms for the detection of code defects and errors have always been of the essence. We can therefore conclude that the ability of the proposed integrated model to perform software testing with a very high degree of accuracy connotes its usefulness to the software engineering industries. Its adoption or integration in the real-world software engineering industries will lead to delay-free development of software. It will also lead to software with very high reliability, resource efficiency, robustness to data, logic, and operational adversities, and good performance metrics.

The false negative values presented in Table 7 indicate the percentages of cases in which the test fails to detect an existing bug or defect in the random samples. In these cases, the software may suffer reliability issues as it may erroneously allow some bugs to slip past testing, which may result in crashes, data loss, security breaches, and user frustration. However, compared to other research outcomes, the false negative values presented in Table 7 fall within a condonable range and establish that the likelihood of the reliability or security issues is minimal.

5. Conclusion

The proposed model (a combination of Random Forest and Adaboost) was trained several times, and the errors obtained were used to update the sample

weights for the next prediction by the Adaboost. After several training the model was tested, and the results obtained were compared with results for the Random Forest. The accuracy prediction of the proposed model was 85.5%, while Random Forest was 80%. However, the proposed model gave a 5.5% improvement in accuracy when compared to the Random Forest model. The significant improvement recorded in accuracy was as a result of using Adaboost, which assigns more weights to weak learners and less weight to strong learners repeatedly until more optimal accuracy is achieved. However, the proposed model can optimally and accurately predict and give promising results. In conclusion, the proposed model was able to give a better and more accurate result because it was able to minimize overfitting, bias, and variance, which can lead to generalization failure.

References

[1] Aggarwal, A., Kumar, S., and Gupta, R. "Testing Coverage-Based NHPP Software Reliability Growth Modelling with Testing Effort and Change-Point". *International Journal of System Assurance Engineering Management,* vol. 15, pp.5157–5166, 2024. https://doi.org/10.1007/s13198-024-02504-7

[2] Pressman, R. S. and Maxim, B. R. "*Software engineering: A Practitioner's Approach*", McGraw-Hill International, 10th edition, McGraw-Hill Publisher, Boston, USA, 2015.

[3] dos Santos, J., Martins, L. E. G., de Santiago Júnior, V. A., Povoa, L. V. and dos Santos, L. B. R. "Software Requirements Testing Approaches: A Systematic Literature Review", *Requirements Engineering,* vol. 25, pp. 317–337, 2020. https://doi.org/10.1007/s00766-019-00325-w

[4] Ammann, P. and Offutt, J. "*Introduction to Software Testing"*. Cambridge University Press 40 W. 20 St. New York, United States, 344, 2008.

[5] Dhore, P., Wadhwa, L., Shinde, P., Chaudhri, D., and Vyas, P. "Brief Review on Different Manual Software Testing Approaches and Procedures," *Journal of Pharmaceutical Negative Results*, pp. 455-464, 2023. https://doi.org/10.47750/pnr.2023.14.S02.56

[6] Taipale, O., Karhu, K. and Smolander, K. "Observing Software Testing Practice from the Viewpoint of Organisations and Knowledge Management", *First International Symposium on Empirical Software Engineering and Measurement* (ESEM 2007), Madrid, Spain, pp. 21-30, 2007. https://www.researchgate.net/publication/4279005_Observing_Software_Testing_Practice_from_the_Viewpoint_of_Organizations_and_Knowledge_Management

[7] Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K., and Eldh, S. "Exploring the Industry's Challenges in Software Testing: An empirical study", *Journal of Software: Evolution and Process*, vol. 32, no 8, 2020. https://doi.org/10.1002/smr.2251

[8] Ateşoğulları, D. and Mishra, A. "Automation Testing Tools: A Comparative View", *International Journal of Information and Computer Security*, vol. 12, no. 4, pp.63-76, 2020

[9] Haas, R., Elsner, D., Juergens, E., Pretschner, A., and Apel, S. "How Can Manual Testing Processes be Optimized? Developer Survey, Optimization Guidelines, and Case Studies. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1281–1291, 2021. https://doi.org/10.1145/3468264.3473922

[10] Sharma, P. "Automated Software Testing Using Metahurestic Technique Based on Improved Ant Algorithms for Software Testing", *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, no. 11, pp.3505–3510, 2015. https://ijritcc.org/index.php/ijritcc/article/view/3497

[11] Bathla, R. and Bathla, S. "Innovative Approaches of Automated Tools in Software Testing & Current Technology as Compared to Manual Testing", *Global Journal of Enterprise Information Systems*, vol. 1, pp.127-131, 2009.

[12] Sabev, P. S. and Grigorova, K. "Manual to Automated Testing: An Effort-Based Approach for Determining the Priority of Software Test Automation", *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 9, no 12, pp.2123-2129, 2015.

[13] Umar, M. A. and Chen, Z. "*A Study of Automated Software Testing: Automation Tools and Frameworks"*, International Journal of Computer Science Engineering, vol. 8, no 6, pp. 217-225, 2019. http://www.ijcse.net/docs/IJCSE19-08-06-011.pdf

[14] Durelli, V. H. S., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., and Dias, D. R. C. "Machine Learning Applied to Software Testing: A Systematic Mapping Study". *IEEE Transactions on Reliability*, vol. 68, no. 3, pp.1189-1212, 2019.

[15] Yadav, D. K. and Dutta, S. "*Regression Test Case Prioritization Technique Using Genetic Algorithm"*. In: Sahana, S.K., Saha, S.K. (eds) Advances in Computational Intelligence. ICCI 2015. Advances in Intelligent Systems and Computing, vol 509. Springer, Singapore, 2017. https://doi.org/10.1007/978-981-10-2525-9_13

[16] Ahmed, F. S., Majeed, A., and Khan, T. "Value–based Test Case Prioritization for Regression Testing Using a Genetic Algorithm", *International Journal of Artificial Intelligence*, vol. 74, no 1, pp.2211–2238, 2023. https://doi.org/10.32604/cmc.2023.032664

[17] Raamesh, L., Jothi, S., and Radhika, S. "Test Case Minimization and Prioritization for Regression Testing using SBLA-Based Adaboost Convolutional Neural Network", *Journal of Supercomputing*, vol. 78, pp.18379–18403, 2022. https://doi.org/10.1007/s11227-022-04540-1

[18] Harikarthik, S. K., Palanisamy, V. and Ramanathan, P. "Optimal Test Suite Selection in Regression Testing with Testcase Prioritization using Modified Ann and Whale Optimization Algorithm", *Cluster Computing*, vol. 22, no 5, pp.11425–11434, 2019. https://doi.org/10.1007/s10586-017-1401-7

[19] Soe, Y. N., Santosa, P. I., and Hartanto, R. "Software Defect Prediction Using Random Forest Algorithm". *Proceedings of the 12th South East Asian Technical University Consortium*, Yogyakarta, Indonesia, pp. 1-5, 2018. https://doi.10.1109/SEATUC.2018.8788881

[20] Singh, A., Katyal, D., and Gupta, D. "Test Case Minimization for Object-Oriented Testing Using Random Forest Algorithm". *Proceedings of the International Conference on Computer Networks, Big Data and IoT*, Springer, vol 49, 2019. https://doi.org/10.1007/978-3-030-43192-1_90

[21] Rathore, S.S. and Kumar, S. "A Study on Software Fault Prediction Techniques", *Artificial Intelligence Review*, vol. 51, pp.255–327, 2019. https://doi.org/10.1007/s10462-017-9563-5