SN Computer Science (2024) 5:530
https://doi.org/10.1007/s42979-024-02764-x

ORIGINAL RESEARCH

An Improved and Optimized Random Forest Based Approach to Predict the Software Faults

Nikhil Saji Thomas[1] · S. Kaliraj[2]

Received: 21 March 2023 / Accepted: 2 March 2024

Abstract

Effective software fault prediction is crucial for minimizing errors during software development and preventing subsequent failures. This research introduces an enhanced Random Forest-based approach for predicting software faults, specifically focusing on the NASA JM1 dataset. The dataset comprises 21 software metrics indicating the presence or absence of faults in a module, and it is utilized to evaluate the proposed approach. The study delves into the intricacies of the NASA dataset, detailing the cleaning process and addressing class imbalance through Synthetic Minority Over-sampling Technique (SMOTE). The core of our approach involves the implementation and fine-tuning of the Random Forest classifier, with a specific focus on optimizing hyperparameters to enhance predictive accuracy. In comparative evaluations with standard machine learning models, our proposed approach demonstrated superior performance, achieving an accuracy of 82.96% and an F1 score of 89.53%. Notably, we emphasize the significance of software defects and their potential to cause failures and crashes during software development, leading to substantial organizational losses. The paper provides a comprehensive examination of different aspects of the machine learning model, offering detailed insights, examples, and illustrative figures to enhance the understanding of our proposed approach.

Keywords Software Fault Prediction · Random Forest, Optimization · NASA Defect Data · Class Imbalance · SMOTE

Introduction

Software fault prediction is the process of identifying potential faults in software systems before they occur. It is an important area of software engineering research that can help to improve the quality and reliability of software systems. In software development, faults can lead to software failures and system downtime, which can have serious consequences [1]. For example, a fault in a financial software system could lead to incorrect calculations or unauthorized access to sensitive information, which can result in financial losses or legal consequences. In some cases, software faults can even lead to injuries or loss of life, such as in the case of software used in medical devices or aviation systems [2].

Software defects have had various impacts in our world, for instance, the year 2000 problem [3], where dates only show the last 2 digits or units of a year. If the year were 2000, the same would not be differentiable from the year 1900, as both show the year as '00'. Tech defects have made their way through the twenty-first century. Pacemakers have been used on people with heart issues. They are used to make sure the heart beats in a stable rhythm. If your pacemaker fails, you are at an increased risk of stroke and heart failure [4]. The healthcare company Abbott, who built their pacemakers to be radio-controlled, and did not add any security features to the pacemakers, made it vulnerable for hackers to hack the patient's heartbeat, increase the heartbeat or turn off the pacemaker. Abbott had to call back all 465,000 who had one to come back to their practice and update their software. These are just very few examples of what tech defects can cause. There are thousands of other tech defects which have been banned or not used now due to various reasons.

✉ S. Kaliraj
kaliraj.s@manipal.edu

Nikhil Saji Thomas
im.nikhilst@gmail.com

[1] Department of Computer Science and Engineering, Manipal Academy of Higher Education, Dubai, United Arab Emirates

[2] Department of Information and Communication Technology, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal 576104, India

Published online: 09 May 2024

By identifying potential faults before they occur, software engineers can take proactive steps to address them and prevent software failures [5, 6]. This can include improving code quality, increasing test coverage, or optimizing software performance. By doing so, software engineers can ensure that software systems are reliable, secure, and meet users' needs.

Software fault prediction researchers faced various challenges in developing accurate and reliable predictive models [7]. Some of the most common problems include.

- Lack of Data.
- Feature selection.
- Model overfitting.
- Imbalanced Datasets.
- Lack of interpretability.

A solution to the Lack of data faced by previous researchers was by using techniques such as synthetic data generation, transfer learning, and active learning [8]. To address the problem of feature selection, researchers used many techniques like wrapper methods, filter methods, and embedded methods, where wrapper methods involve evaluating the performance of the model with a different subset of features, while filter methods involve selecting features based on statistical measures such as correlation or mutual information. Embedded methods involve selecting features as part of the model training process. To address model overfitting, researchers can use techniques such as regularization, cross-validation and early stopping [9]. Imbalanced data can be treated using techniques like oversampling, under sampling and cost-sensitive learning. An example of oversampling is SMOTE (Synthetic Minority Oversampling Technique) [10], which has been used in our model. Black-box models such as neural networks can be difficult to interpret, which is a problem in safety–critical applications. To address this problem, researchers can use explainable AI techniques such as decision trees, rule-based models, etc.

The algorithm we use is random forest classification with hyperparameter tuning with cross-validation. There are various other classification techniques that can be used instead of the random forest classification we have used in this model. There's logistic regression, decision tree, Naïve Bayes, etc. What makes Random Forest stand out from other classification techniques? They stand out from many other classification algorithms because of its ability to reduce overfitting, handle missing data, and work well with high-dimensional datasets [11].

Random forest combines multiple decision trees [12], which helps reduce overfitting and increase accuracy. It can work well with high-dimensional datasets, which means it can handle many input features. It can handle missing data without much pre-processing, unlike some other algorithms. It can provide feature importance scores, which can help you understand which features are most important for classification.

Due to the above reasons, the results delivered by using Random Forest compared to other classification techniques are comparatively better.

The data sets used are Nasa's defective data set. They are observations taken from software built by NASA, where each of the feature columns is each feature reading, and the result of the reading is the target/label column defects.

Despite the strides made in software fault prediction, there remains an ongoing need for robust and efficient models. This paper addresses this need by introducing an advanced Random Forest-based approach tailored for the unique intricacies of software fault prediction. The motivation behind our work is to not only contribute to the growing body of knowledge in this field but, more crucially, to provide the software engineering community with a tool that surpasses existing models in terms of accuracy and reliability. By leveraging the distinctive capabilities of Random Forest, our proposed approach aims to mitigate common challenges faced by predictive models, such as overfitting and handling high-dimensional datasets. The subsequent sections delve into the intricacies of our proposed model, offering insights into its implementation, comparison with standard models, and a thorough analysis of results. This work stands as a testament to our commitment to advancing the field of software fault prediction and providing practitioners with a dependable tool for enhancing the reliability and security of software system.

The following section of the paper has been organized as mentioned. Section "Literature Review" discusses about the background of software fault prediction using machine learning and deep learning techniques. The proposed approach and its process are explained in Section "Proposed Approach and its Implementation". The detailed steps to implement the proposed work are also discussed in Section "Proposed Approach and its Implementation", and this section has the explanation with its source code wherever required, to understand the process clearly. The proposed improved random forest classifier has been compared with other standard models, and the results are discussed in Section "Result Analysis", followed by the conclusion of the work is given.

Literature Review

Machine learning has been used several times widely for software vulnerability prediction. Software vulnerability prediction has been a broad area of research, and machine-learning models have been made using various methods.

For example, an early approach to it, An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics [13], such as McCabe's cyclomatic complexity, nesting complexity, and size. They used binary Logistic Regression for analysis. Later, this method was modified to bring up more advanced machine learning models like [14] A research by Y. Shin and A. Meneely, which says, "Another way of identifying the most vulnerable code locations is to use process characteristics of code development. Perhaps code that has undergone recent changes (i.e. code churn) might have new vulnerabilities". By the time gap, they advanced the earlier model by adding code churning, ie; code that is rewritten or deleted shortly after being written and other key aspects which helped understand vulnerability and help in predicting threats). Let's look at a few Defect Prediction machine learning models this way.

A machine learning (ML)-based technique to software bug prediction was presented by Hammouri et al. in 2018 [15]. Based on past data, three monitoring ML approaches were utilised to forecast potential software faults. The evaluation technique indicated that ML algorithms can be employed appropriately and effectively. Empirical findings indicated that the ML methodology outperforms other strategies, such as linear AR and POWM models, in terms of efficacy for the estimate method.

The collective sorting method-based strategies for identifying software bug faults were examined by Moustafa et al. in [16]. The techniques were assessed using datasets of varying sizes and used to incorporate different groups of software measures as elements of the sorting algorithms. The results showed that update measurements outperformed static code measurements and an equal parts data combining approach.

Adam A. Porter and Richard W. Selby [17] had used a classification tree using method-level metrics. Metric-based classification trees can provide an empirically guided approach for identifying various classes of high-risk software components throughout the software life cycle. The metric-based classification tree approach has been validated using NASA and Hughes project data. The model built had 79.3% accuracy, as imbalanced datasets cannot be evaluated with metric, and accuracy metric is not applicable for software fault prediction studies.

In a study done by Luca P, Fabio P and Alberto B [18], different classifiers were used on a software defect prediction machine learning model like binary logistic regression, ADTree, multilayer perceptron, naive Bayes, and random forest. They used the random forest algorithm as it gave a higher performance to 10 data sets. The result of short-term defect prediction was proved to be effective.

A paper done by Lipika G, Mayanak S and Sunil KK [19] shows the usage of synthetic minority over-c technique (SMOTE) to preprocess the datasets focusing on the severe imbalance problem of data sources from different projects because there exists an imbalance between the defect prone and non-defect prone classes.

Kishan K G and B M Mainul H [20] used a NASA defect data set. They preprocessed the data sets with repeated attribute removal and noise removal using machine learning algorithms. Cleaning the data improved the performance of learning. Results showed that learning algorithm used, like Random Forest, Naïve Bayes algorithm have an appealing performance.

The issue faced by Adam A. Porter [17] with the NASA and Hughes project was the imbalanced data. Imbalanced data sets could later be passed through SMOTE (synthetic minority over-c technique) Like we see in the study done by Lipika G, Mayanak S and Sunil KK [19]. SMOTE is the same technique used in our NASA defect prediction model to balance the number of TRUE and FALSE values. The Random Forest classifier shows the best results in many machine learning models. The cleaned NASA software defect data set we used was passed through different classifiers, random forest showing the best results. Referring [20] shows cleaning a NASA defect data set and [18] shows Random Forest showing the best results for their data set compared to few other classifiers used.

Singh [21] used exploratory data analysis to generate the initial rule base, in particular, the k-means clustering algorithm. Then they converted each cluster into a fuzzy rule. The results of their experiments demonstrate that their proposed fuzzy rule-based classification approach can yield competitive or better performance than C4.5, random forest learner, and Naive Bayes classifier.

By hyperparameter tuning the Random Forest Classifier, we can get a better accuracy rate and outcome. There are various types of hyperparameters that can be tuned in the random forest, like grid search, randomized grid search, etc. Readers can refer to a study by Philipp Probst, Marvin Wright and Anne-Laure Boulesteix [22], which describes hyperparameters and the different strategies that can be used for tuning Random Forest Classifier.

We found that many researchers concentrated on machine learning algorithm-based prediction and deep learning approaches when we analyzed the literature review. Most researchers in this field process all the characteristics directly rather than concentrating on the best ones otherwise, machine learning algorithms were not optimized to get better accuracy in fault prediction. The complexity and time required for computation will rise as a result. This research suggests feature selection-based software bug prediction using an optimized random forest algorithm. Our proposed approach uses randomized grid search and hyperparameter tuning to optimize the performance of the random forest algorithm to get better accuracy in software defect prediction.

Proposed Approach and its Implementation

The conceptual framework for software fault prediction is visually represented in Fig. 1. The process initiates with the preprocessing of the initial dataset, where efforts are made to address null values and rectify other data-related issues. Subsequently, Synthetic Minority Over-sampling Technique (SMOTE) is applied to rectify class imbalances, a critical step that enhances the robustness of the predictive model during training. Following this, the Random Forest feature selection mechanism is employed to meticulously curate a set of features that optimally contribute to the predictive accuracy. The core of our methodology lies in the utilization of the Random Forest classifier, a versatile algorithm chosen for its capacity to effectively handle high-dimensional datasets and mitigate overfitting. To enhance the performance of the Random Forest classifier, a two-fold optimization process is implemented, combining randomized grid search and hyperparameter tuning. This fine-tuned classifier is then employed to predict software faults with a heightened level of accuracy. The subsequent sections provide an in-depth exploration of each step in this proposed methodology, offering a comprehensive understanding of its theoretical foundations and practical implementation.

Preprocessing the Dataset

The NASA JM1 dataset comprises 2631 rows and 22 columns, with one column serving as the label or target variable. Upon importing the data, a preliminary analysis was conducted using the df.describe() function, providing a statistical overview showcased in Fig. 2.

To gain a detailed insight into the dataset's features, including counts and data types, df.info() was employed. The result of this function is depicted in Fig. 3.

Notably, certain features, such as uniq_Op, uniq_Opnd, total_Op, total_Opnd, and branchCount, exhibit an 'object' data type. To circumvent potential issues arising from this during data imbalance treatment with SMOTE, a necessary transformation was implemented. The 'object' data type was converted to 'float', as illustrated in Fig. 4.

Additionally, the dataset presented instances of Null values, denoted as NaN or Na. To ensure the integrity of subsequent analyses and model training, a systematic approach for handling these Null values was employed, as elucidated in Fig. 5.

Solving Class Imbalance Issue

Class imbalance can significantly impact the performance of predictive models, particularly in scenarios where one class

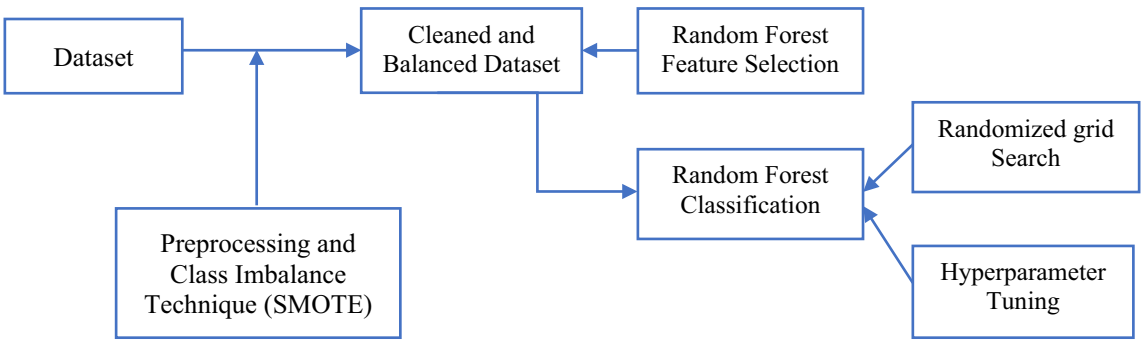


Fig. 1 Proposed approach for fault prediction

```
df.describe()
```

	loc	v(g)	ev(g)	iv(g)	n	v	l	d	i	e	b	t	lOCode	lOComment	lOBlank	locCodeAndComment
count	2631.000000	2631.000000	2631.000000	2631.000000	2631.000000	2631.000000	2631.000000	2631.000000	2630.000000	2.630000e+03	2630.000000	2.630000e+03	2630.000000	2630.000000	2630.000000	2630.000000
mean	72.434854	10.645154	5.282174	6.604485	196.548195	1264.735177	0.088856	19.301311	40.165494	9.571548e+04	0.422529	5.317527e+03	45.097719	5.115589	7.839163	0.715589
std	130.598471	22.923653	10.088760	16.406490	414.728714	3426.613111	0.118460	25.888198	48.724881	8.561854e+05	1.142542	4.756585e+04	105.059225	13.947612	15.725331	3.083179
min	1.000000	1.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000e+00	0.000000	0.000000e+00	0.000000	0.000000	0.000000	0.000000
25%	19.000000	2.000000	1.000000	2.000000	22.000000	83.395000	0.020000	4.500000	14.555000	3.985325e+02	0.030000	2.214500e+01	6.000000	0.000000	1.000000	0.000000
50%	39.000000	5.000000	1.000000	3.000000	82.000000	404.170000	0.050000	12.830000	27.940000	5.480525e+03	0.135000	3.044750e+02	20.000000	0.000000	4.000000	0.000000
75%	84.500000	11.000000	6.000000	6.000000	205.000000	1150.450000	0.110000	26.565000	49.015000	3.058646e+04	0.380000	1.699250e+03	49.000000	5.000000	9.000000	0.000000
max	3442.000000	470.000000	140.000000	402.000000	8441.000000	80843.080000	1.300000	408.730000	569.780000	3.107978e+07	26.950000	1.726655e+06	2824.000000	344.000000	447.000000	108.000000

Fig. 2 Data Analysis—output of describe() function

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2631 entries, 0 to 2630
Data columns (total 22 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   loc                 2631 non-null   float64
 1   v(g)                2631 non-null   float64
 2   ev(g)               2631 non-null   float64
 3   iv(g)               2631 non-null   float64
 4   n                   2631 non-null   float64
 5   v                   2631 non-null   float64
 6   l                   2631 non-null   float64
 7   d                   2631 non-null   float64
 8   i                   2630 non-null   float64
 9   e                   2630 non-null   float64
 10  b                   2630 non-null   float64
 11  t                   2630 non-null   float64
 12  lOCode              2630 non-null   float64
 13  lOComment           2630 non-null   float64
 14  lOBlank             2630 non-null   float64
 15  locCodeAndComment   2630 non-null   float64
 16  uniq_Op             2630 non-null   object
 17  uniq_Opnd           2630 non-null   object
 18  total_Op            2630 non-null   object
 19  total_Opnd          2630 non-null   object
 20  branchCount         2630 non-null   object
 21  defects             2630 non-null   object
dtypes: float64(16), object(6)
memory usage: 452.3+ KB
```

Fig. 3 Data Analysis—output of info() function

is heavily outnumbered by the other. In our case, addressing this imbalance becomes imperative to ensure the classifier's ability to discern both defective and non-defective modules effectively (see Fig. 6).

To combat this issue, we employ the Synthetic Minority Over-sampling Technique (SMOTE), a method rooted in data augmentation principles. SMOTE strategically generates synthetic instances of the minority class, enhancing its representation in the dataset. This process aims to create a more balanced distribution of True and False values within the 'defects' column, fostering a more equitable learning environment for the subsequent stages of model training. The integration of SMOTE not only mitigates class imbalance but also contributes to the overall robustness of the predictive model, ensuring a more comprehensive exploration of the feature space during training.

Steps to Solve the Class Imbalance Problem

The data set we are using is imbalanced and binary classification. Since the data sets are imbalanced, we use SMOTE, which is a synthetic minority oversampling technique. This is one of the most used oversampling methods to solve the imbalance problem. It aims to balance class distribution by randomly increasing minority class examples by replicating them.

Step 1: Setting the minority class set A, for each $x \epsilon A$, the k-nearest neighbors of x are obtained by calculating the Euclidean distance between x and every other sample in set A.

Step 2: The sampling rate N is set according to the imbalanced proportion. For each $x \epsilon A$, N examples (i.e. $\times 1, \times 2, \ldots$xn) are randomly selected from its k-nearest neighbors, and they construct the set A 1.

Step 3: For each example $x_k \in A$ 1(k = 1, 2, 3…N), the following formula 1 is used to generate a new example:

$$x\prime \in x + rand(0, 1) * |x - x_k| \quad (1)$$

in which rand(0, 1) represents the random number between 0 and 1.

Observation:

X_train before smote: 21 features, 1841 rows × 21 columns

X_train after smote: 21 features, 2974 × 21 columns

y_train before smote: 1841 rows × 1 column

True: 2106

False: 524

y_train after smote: 2973 rows × 1 column

True: 1487

False: 1487

The results shown in Fig. 7 are after applying SMOTE technique in the dataset.

Fault Prediction Using Optimized Random Forest Classifier

In the pursuit of an optimal machine-learning model to predict the label or target column, we conducted a thorough exploration of various classifiers within our dataset. The accuracy of these classifiers is summarized in Table 1.

Fig. 4 Process—The object data type to float data type

```
df['uniq_Op'] = pd.factorize(df['uniq_Op'])[0].astype(float)
df['uniq_Opnd'] = pd.factorize(df['uniq_Opnd'])[0].astype(float)
df['total_Op'] = pd.factorize(df['total_Op'])[0].astype(float)
df['total_Opnd'] = pd.factorize(df['total_Opnd'])[0].astype(float)
df['branchCount'] = pd.factorize(df['branchCount'])[0].astype(float)
```

Fig. 5 Code to remove null values

```
[ ] df.replace([np.inf, -np.inf], np.nan, inplace=True)

[ ] nullvalues=df.isna().sum().to_frame()
    for index, row in nullvalues.iterrows():
        print(index, row[0])

    loc 0
    v(g) 0
    ev(g) 0
    iv(g) 0
    n 0
    v 0
    l 0
    d 0
    i 1
    e 1
    b 1
    t 1
    lOCode 1
    lOComment 1
    lOBlank 1
    locCodeAndComment 1
    uniq_Op 1
    uniq_Opnd 1
    total_Op 1
    total_Opnd 1
    branchCount 1
    defects 1

[ ] df.dropna(inplace=True)
```

```
df['defects'].value_counts()

True     2106
False     524
Name: defects, dtype: int64
```

Fig. 6 Count of defective and non-defective modules in the dataset

```
y_res.value_counts()

1    1487
0    1487
Name: defects, dtype: int64
```

Fig. 7 Balanced classes in the dataset after SMOTE technique

Table 1 Fault prediction accuracy of Standard Classifiers

S. no	Classifier	Accuracy (%)
1	Logistic Regression	45
2	Naïve Bayes	55.8
3	K-Nearest Neighbors	52.6
4	Random Forest	72.75

The observed accuracy values reveal Random Forest as the standout performer among the considered classifiers. Random Forest's distinctive approach involves leveraging multiple decision trees, each trained on different subsets of the data through bootstrapping. The final predictions are then aggregated, a technique known as bagging, to yield a more robust and accurate outcome.

The superior accuracy demonstrated by the Random Forest classifier underscores its ability to handle the complexity of the feature space effectively. The ensemble nature of Random Forest contributes to its resilience against overfitting, making it particularly advantageous for datasets with diverse

Table 2 Sample original dataset

id	X0	X1	X2	y
0	3.5	6.1	5.9	0
1	6.5	2.9	4.7	1

Table 3 Newly created dataset

id	X0	X1	X2	y
0	6.5	4.1	5.9	0
1	6.5	2.9	4.7	1

and high-dimensional features. The subsequent sections delve into the optimization process of the Random Forest classifier, where randomized grid search and hyperparameter tuning play pivotal roles in fine-tuning the model for enhanced predictive performance.

Random Forest Classifier

Random Forest is Classifier used in Machine Learning. It is a classification algorithm containing many decision trees. A decision tree splits a data set recursively using the decision modes unless we are left with a pure leaf node, and it finds the best split by maximizing the entropy gain. A small modification in the data set can cause changes in the decision tree. This shows that decision trees are highly sensitive to the training data. Our model might fail to generalize due to inconsistency or high variance. For example, the original sample dataset, the newly created datasetand its decision tree is shown in Tables 2, 3 and Fig. 8.

Hence, we use Random Forest classifier. Random Forest is a collection of multiple random decision trees. It is much less sensitive to the training data. There are multiple decision trees in this classifier, hence the name **Forest**.

Process in a Random Forest Random forest algorithm builds new data sets from the original data set, taking a same number of rows. There will be repeating rows in the newly created data sets, this is because it's performing random sampling with replacement, ie; after selecting a row, it goes back into the data. The process we follow to create new sets of data from the original data set is known as Bootstrapping. Consider, An original data contains 5 ids' namely 0, 1, 2, 3, 4, and from this let's take 4 newly created data sets as an example which is shown in Fig. 9.

Now the algorithm trains a decision tree on each of the bootstrapped data sets. The algorithm won't choose all the features for training the trees. It randomly selects a subset of features for each of the trees and uses only them for training.

From the newly created data sets, decision trees are created, and a series of test sets are passed through each decision tree to give an output. These output values from each tree is taken and combined. Since it is a classification, the algorithm takes the combination of the prediction and gives the majority

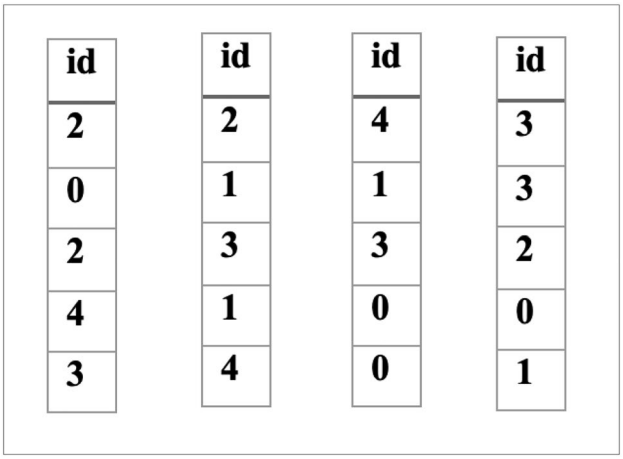


Fig. 9 Considered newly created dataset

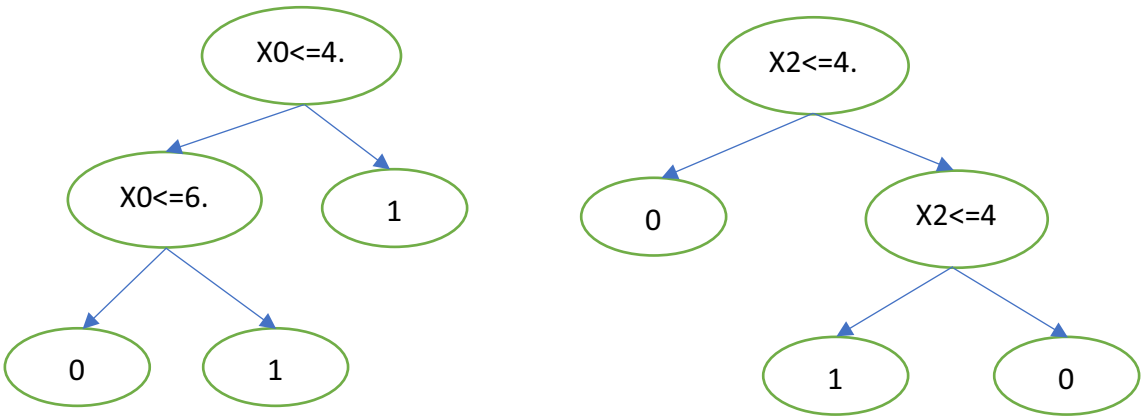


Fig. 8 Random Forest Decision Tree

prediction as the output. This technique of creating new data sets and combining their prediction is known as bagging, which is shown in formula 2. Bagging uses a voting classifier, ie; the majority votes will be the output.

$$\text{Bagging} = \text{Bootstrapping} + \text{Aggregation} \tag{2}$$

Why Bootstrapping and Random Feature Selection? Bootstrapping ensures that we are not using the same data for every tree, so in a way it will help our model to be less sensitive to the original training data. Boot strapping keeps on choosing random data from the dataset till it meets the same number of rows on the actual data set. The allowing of Duplicates by bootstrap is known as sampling with replacement.

The random forest feature selection helps to reduce the correlation between the trees. If we use every feature then most of the trees will have the same decision nodes and they will act very similar, which can increase the variance.

Decision Trees As mentioned earlier, Random Forest classifier is a collection of decision trees. Now, what are decision trees, and how do they work? Decision trees are simple to understand. Let's take a look at a simple structure of decision tree shown in Fig. 10.

The data which do not meet the condition will go to the right, and the one that meets go to the left. The computer decides the decision split, which gives the minimum information gain. We use entropy to find the child nodes. Entropy is a measure of information contained in a state. The entropy calculation is given in formula 3.

$$Entropy = \sum -pi \log(pi) \tag{3}$$

where, pi = probability of class i.

If entropy is high, then we are very uncertain about the randomly picked data, and we need more bits to describe this state. Let's take another instance, if we have an equal probability of 1's and 0's, then the pi is 0.5. So then the entropy corresponding to the root state is 1, which is the highest possible value of entropy.

As shown in Fig. 11, the state with 4 red points gives the minimum entropy, that's why we call it a pure node. Now, to find the information gain corresponding to a split as given in formula 4, we need to subtract the combined entropy of the child node from the entropy of the parent node.

$$\text{IG} = \text{E(parent)} - \sum wi\, E(childi) \tag{4}$$

$$IG_1 = 1 - \frac{14}{20} \times 0.99 - \frac{6}{20} \times 0.91 = 0.034 \tag{5}$$

$$IG_1 = 1 - \frac{4}{20} \times 0 - \frac{16}{20} \times 0.95 = 0.24 \tag{6}$$

From the example, the second split gives us greater information gain as given in the calculation 5 and 6 ($IG_1 < IG_2$). So we choose Decision 2 for split. The model

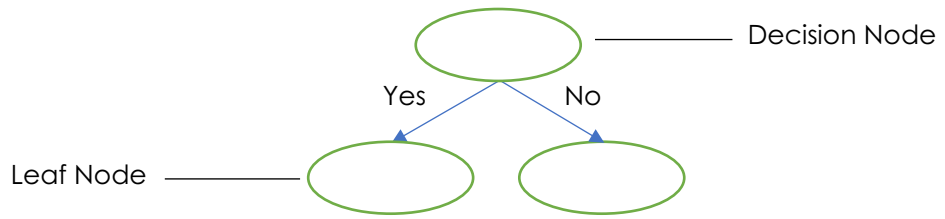


Fig. 10 Simple structure of Decision Tree

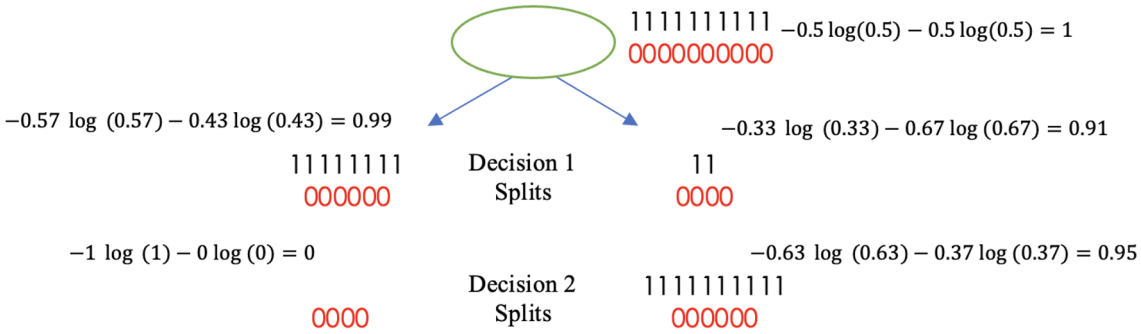


Fig. 11 Decision split based on entropy

compares every possible split in reality and takes the one with maximum information gain. The model traverses through every feature and feature value. Again, what are Decision trees? A decision tree is a greedy algorithm which selects the current best split that maximizes information gain. It does not backtrack, i.e. go back and change a split. The random forest pseudocode and prediction algorithm steps are summarized below.

Random Forest pseudocode:

1. *Building a new data set from the original data set with bootstrapping.*

Randomly select **"k"** *features from total* **"m"** *features.*

Where k < < m

2. *Creates new decision trees for each of the newly created data set*

Among the "k" features, get the node **"d"** *using the best decision split point.*

Splits the nodes using information gain, the decision split which gives the greatest Information gain is used for the split.

$$IG = E(parent) - \sum wi\, E(childi)$$

Split the node into child nodes using the best split.

3. *Repeat steps 1 and 2 until "i" number of nodes are created.*

4. *Build forest by repeating steps 1 to 3 for "n" number times to create a tree.*

Random Forest Predication algorithm:

1. *Randomly selects a subset of features for each of the tree and they are used for training.*

2. *Series of test sets are passed through each decision tree and generate predicted outcomes.*

3. *Voting classifiers are used to check which prediction is in majority to take the final prediction.*

Optimization Process

As per research, the best-known optimization for Random Forest is hyperparameter tuning, Grid Search, Randomized Grid search, Cross Validation. In the Model, we used Randomized Grid Search, which helped the accuracy jump from 72.75 to 80%. By hyperparameter tuning it gives the best accuracy of 82%. Using Random Forest Feature Selection gives an accuracy of 82.96%.

Randomized Grid Search Cross Validation

As we use Random Forest, the best-known optimizer for Random forests are Grid Search, Randomized Grid Search, and Hyper Parameter tuning. We only have a rough idea of the best hyperparameters. Thus the best way to find the optimum hyperparameter and its value is to evaluate a wide range of values for each hyperparameter.

There are default parameters give in Scikit-learn's random forest. As a data scientists our main objective is to increase the accuracy of the model. So, we use hyperparameter tuning to tune the parameters of the given random forest classifier.

The package Randomized grid search helps us to tune our model with hyperparameters. There are various hyperparameters for Random Forest. Some of the important ones are,

1. max_depth: The max_depth of a tree in Random Forest is known as the longest path from the root node to the leaf node.
2. min_sample_split: This parameter tells us the minimum number of observations in any node in order to split it in a decision tree.
3. min_samples_leaf: This hyperparameter tells the minimum number of samples that should be present in the leaf node after splitting a node.
4. n_estimators: n_estimators are the number of trees you want to build before taking the maximum voting or averages of predictions.
5. max_features: This hyperparameter resembles the number of maximum features provided to each tree in a random forest.

These are just a few; there are way more parameters than this, like max_samples, max_terminal_nodes, etc. In randomized grid search, the parameters are tried with different inputs, and several combinations of the parameters are taken to find the best combination.

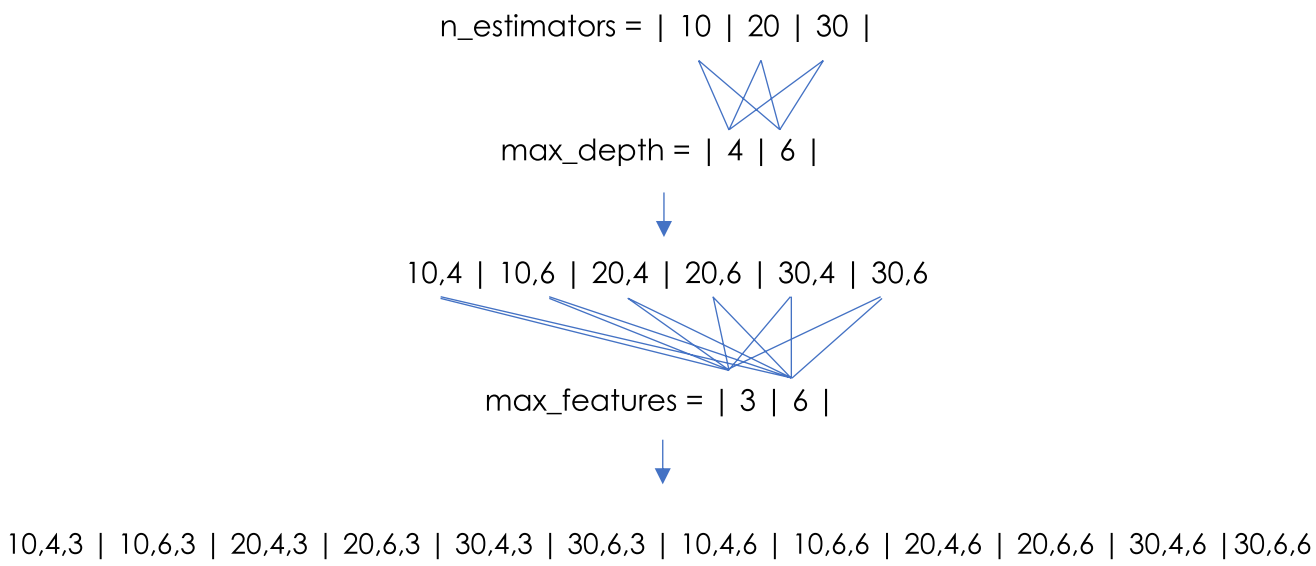
For better understanding, let's take an instance and perform a grid search. Let's take an instance of three parameters and give each of them a set of values to test out.

n_estimators = [10, 20, 30].

max_depth = [4, 6]

max_feature = [3, 6]

The above is our set of input values as an example. Let's see how they combine and form a set of different combinations to find the best optimal combination.



So here the total number of combinations can be calculated using formula 7.

$$\begin{aligned} &\mathbf{3}\text{ values for n\_estimator } \times \mathbf{2}\text{ values for max\_depth} \\ &\quad \times \mathbf{2}\text{ values for max\_features} \\ &= \mathbf{12}\text{ total combinations} \end{aligned} \tag{7}$$

The accuracy of these combinations is then compared with each other to find the best optimal value. Since Grid Search takes a lot of time and space, for large data sets, we use Randomized Grid search. In Randomized Grid Search, it will not run for all combinations, it will only pick values randomly and combine them to form combinations.

What parameters do randomized grid search use? When we run a randomized Grid Search CV, we give a parameter called **n_iter**. This iteration tells how many times the model has to be run. If n_iter is 50 then the model takes 50 randomly selected combinations and if n_iter is 100 then the model will take 100 randomly selected combinations. On what basis does it choose the combination? For this we have to give another parameter which chooses the distribution known as param_distributios. Randomized Grid Search CV have several parameters, n_jobs, verbose are also some of them.

Cross Validation (CV)

Let's take an instance, A student has a doubt on a particular topic and asks that doubt his classmate to get it cleared. The student again goes to his professor and asks the same doubt to make sure what his classmate said is right. Here the student is trying to cross-validate. The same applies in the world of Machine learning; when there is input data, the data is split into train data and test data. The machine trains the model with the training data and tests the model with the test data. It is not possible to take another set or sets of test data again if we need to check on the model again. Increasing the test data can decrease the training data, which is not good for the machine-learning model. Hence, we use CV or cross-validation.

In cross-validation, the original data is split into K number of subsets with a random selection of rows. These subsets fit into the model. The subsets are then split into a training set and a test set in each cross-validation.

For Example, Let's take data with 50 rows, and divide it into 5 subsets as shown in Fig. 12. So, each subset contains 10 random rows in it. Let's call the subsets s1, s2, s3, s4 and s5. Here, the algorithm takes s1, s2, s3 and s4 as the training set and s5 as the test set. In the next cross-validation, the algorithm randomly takes maybe s2, s3, s4, s5 as the training set and s1 as the test set, and so on for each validation n number of times (cv=n).

Hyper Parameter Tuning

Randomized Grid Search CV takes data randomly from the grid, which makes the model faster. Randomized Grid search CV is used for large data sets and gives the best randomly generated combination. Since they are randomly generated, they might have a slightly lower accuracy than it's actual potential. Hence, we can also

Fig. 12 Process of Cross-Validation

try Hyperparameter tuning manually, which can take a very long time, but it might give better accuracy than the hyperparameters chosen by Randomized Grid Search CV.

Hyperparameters are like the setting of an algorithm that can be adjusted to optimize performance. The data scientist must set these hyperparameters before training. In random forest, hyperparameters include the number of decision trees in the forest, and the number of features considered by each tree when splitting a node. The parameters of the random forest are the variables, and the thresholds used to split each node learned during training.

When using random forest, Scikit Learn implements a set of sensible default hyperparameters for all models, but these are not guaranteed to be optimal for a problem. The best parameters are usually impossible to determine ahead of time. By optimizing the model for the training data, our model will score very well on the training set.

We get the best accuracy from Random Forest Classifier. we are optimizing the random forest classifier and hence

```
# Number of features to consider at every split
max_features = [0.3]
# Maximum number of levels in tree
max_depth = [70]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [8]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1]
# Method of selecting samples for training each tree
bootstrap = [True]
```

```
model2 = RandomForestClassifier(oob_score =True, n_estimators=1400)
rf1_random = RandomizedSearchCV(estimator = model2, param_distributions =
                                random_grid, n_iter = 100, cv = 3, verbose=2,
                                random_state=42, n_jobs = -1)
```

Fig. 13 The best combination of Hyperparameters

use Randomized Grid Search, which is one of the best optimizers for Random Forest. By using Randomized Grid Search, we get about 80% accuracy. Now, by using hyperparameter tuning, we could try to obtain even better accuracy.

For this, various combinations of hyperparameters are tried. Figure 13 shows the best combinations we have got.

Setting the max_features between 0.2 and 0.5 gives good accuracy. Hence, keeping it at 0.3 gives a better chance of achieving better accuracy. max_depth of value 70 was showing better results that with max_depth of 100.

With several trials of Randomized Grid Search, the best accuracy that we keep getting is 82.9%. To acquire better accuracy, we tried feature selection. For this, Random Forest feature Selection is used. Even using Random Forest Feature selection gives a maximum accuracy of 82.9%.

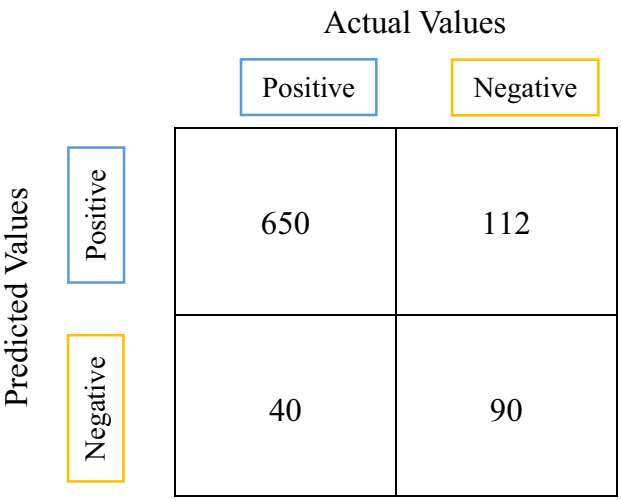


Fig. 15 Proposed approach confusion matrix

Result Analysis

In the result analysis, we delve into a comprehensive examination of the outcomes derived from our proposed methodology. The efficacy of our approach is rigorously assessed, considering key performance metrics such as accuracy, precision, recall, and F1 score. The results are presented and interpreted in light of the specific objectives outlined in the study. Through a meticulous exploration of the obtained results, we aim to provide meaningful insights into the predictive capabilities of the optimized Random Forest classifier and its potential implications for software fault prediction.

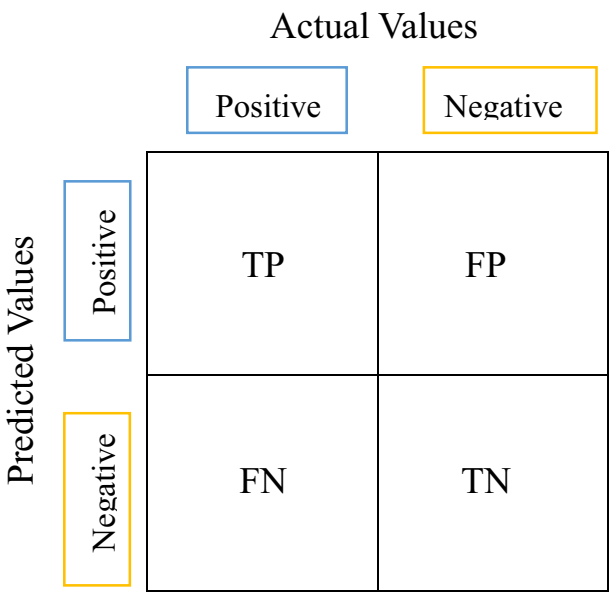


Fig. 14 General Confusion Matrix

The confusion matrix is a performance measurement for machine learning classifications. It is a 2×2 table with predicted and actual values as shown in Fig. 14. They show true positive, false positive, true negative and false negative values. From the confusion matrix, we can derive other readings like Precision, Recall, and F1-Score.

True positive (TP): The actual value is a positive and the predicted value is also a positive.

True negative (TN): The actual value is a negative and the predicted value is also a negative.

False positive (FP): The actual value is a negative but the predicted value is positive.

False negative (FN): The actual value is positive but the predicted value is negative.

In our model with 30% as test set having 892 test data there are 80 True Negatives, 130 False Positives, 50 False Negatives, and 632 True Positives, which is given in Fig. 15.

The accuracy of the model is calculated using formular 8,

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

Accuracy of our model is,
$\frac{650+90}{650+90+112+40} = 0.8296$, That is, 82.96% accuracy.

Recall

$Recall = \frac{TP}{TP+FN}$ And $Recall = \frac{TN}{TN+FP}$

Recall shows from all the positive values how many positive values were correctly predicted. The same goes for the negative values.

In our model, recall for True,

$\frac{650}{650+40} = 0.942$, That is 94.2%

And for our False,

$\frac{90}{90+112} = 0.44$, That is 44%

Precision

$Precision = \frac{TP}{TP+FP}$ And $Precision = \frac{TN}{TN+FN}$

Precision shows from all the predicted positive values how many are actually positive. The same goes for the negative values.

In our model, Precision for True,

$\frac{650}{650+112} = 0.853$, That is 85.3%

And for our False,

$\frac{90}{90+40} = 0.69$, That is 69%

As mentioned earlier in the Proposed Approach section, the followings are the different classifications that have gone through and their accuracy.

Logistic Regression = 47.09%

Naïve Bayes = 55.83%

K-Nearest Neighbors = 52.69%

Random Forest = 72.76%

Proposed Approach = 82.96 %

From the confusion matrix shown in Fig. 16 and the graph shown in Fig. 17, comparing the True Positive and

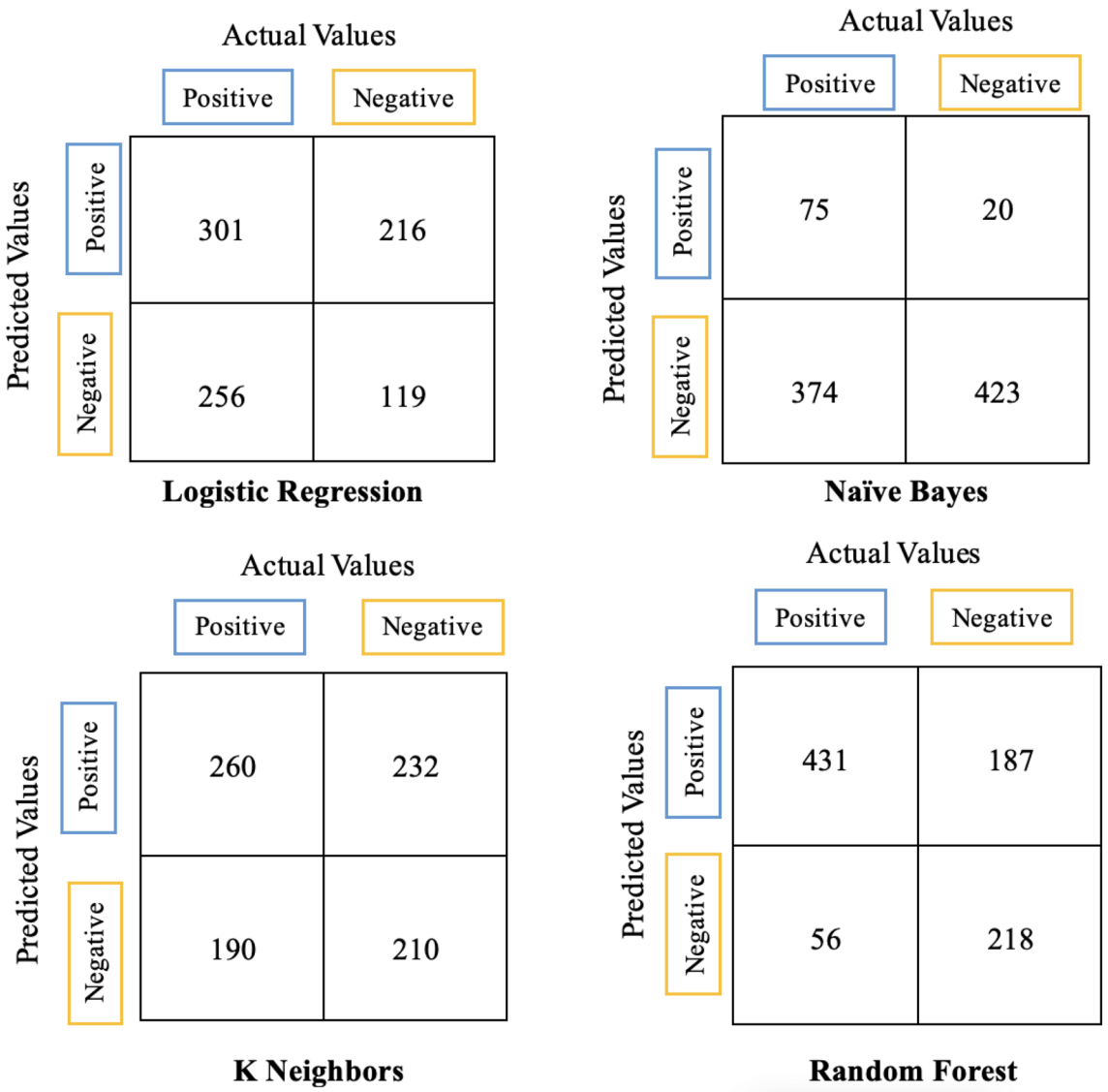


Fig. 16 Confusion matrix of standard classifiers

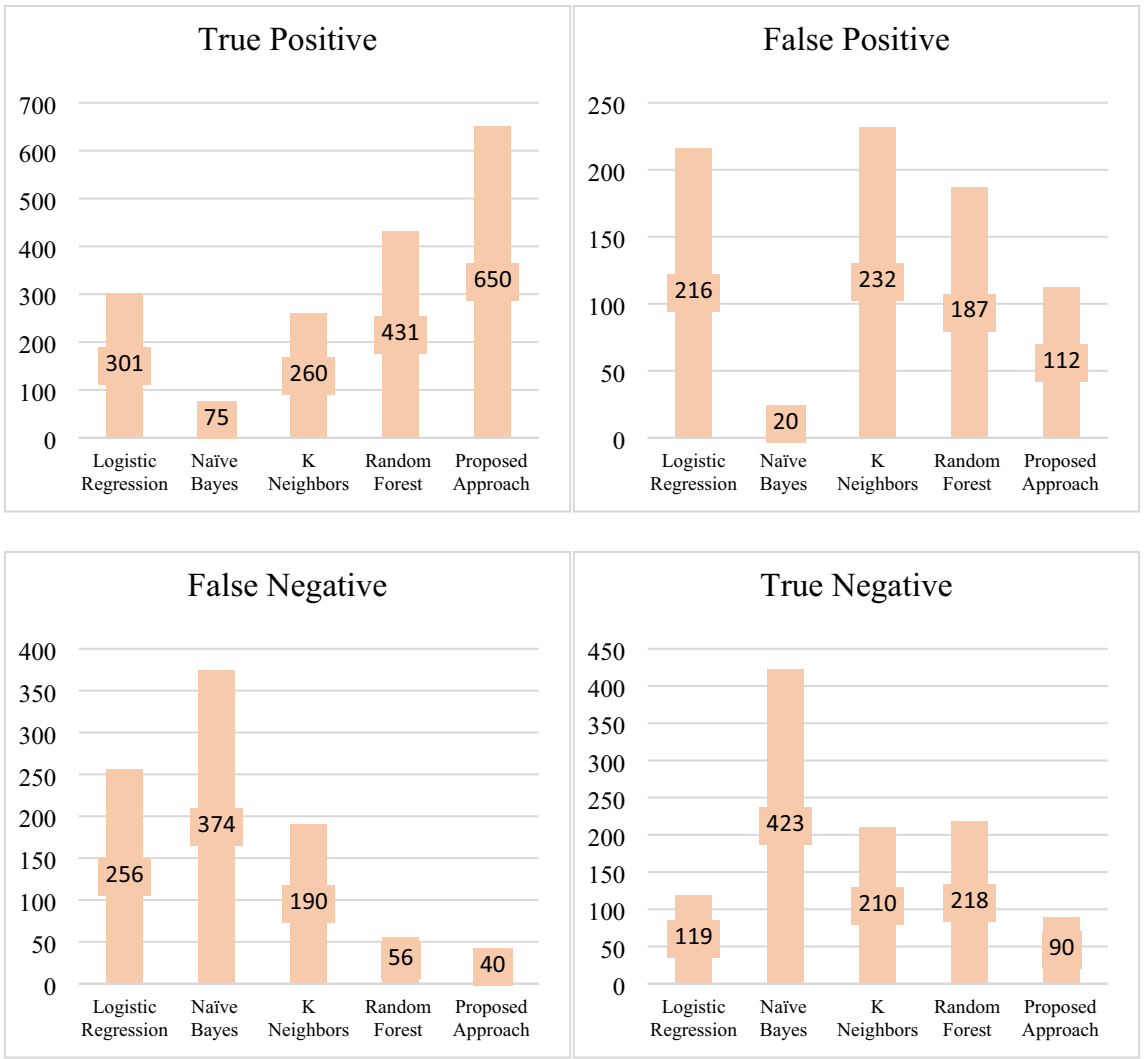


Fig. 17 TP, FP, FN, TN Comparison of standard classifier with proposed approach

Table 4 Performance Measures of Classifiers

Measure	Logistic regression	Naive Bayes	K Neighbours	Radom forest	Proposed approach
Sensitivity	0.5404	0.167	0.5778	0.885	0.942
Specificity	0.3552	0.9549	0.4751	0.5383	0.4455
Precision	0.5822	0.7895	0.5285	0.6974	0.853
Negative Predictive Value	0.3173	0.5307	0.525	0.7956	0.6923
False Positive Rate	0.6448	0.0451	0.5249	0.4617	0.5545
False Discovery Rate	0.4178	0.2105	0.4715	0.3026	0.147
False Negative Rate	0.4596	0.833	0.4222	0.115	0.058
Accuracy	0.4709	0.5583	0.5269	0.7276	0.8296
F1 Score	0.5605	0.2757	0.552	0.7801	0.8953
Matthews Correlation Coefficient	−0.1024	0.1976	0.0532	0.4568	0.4597

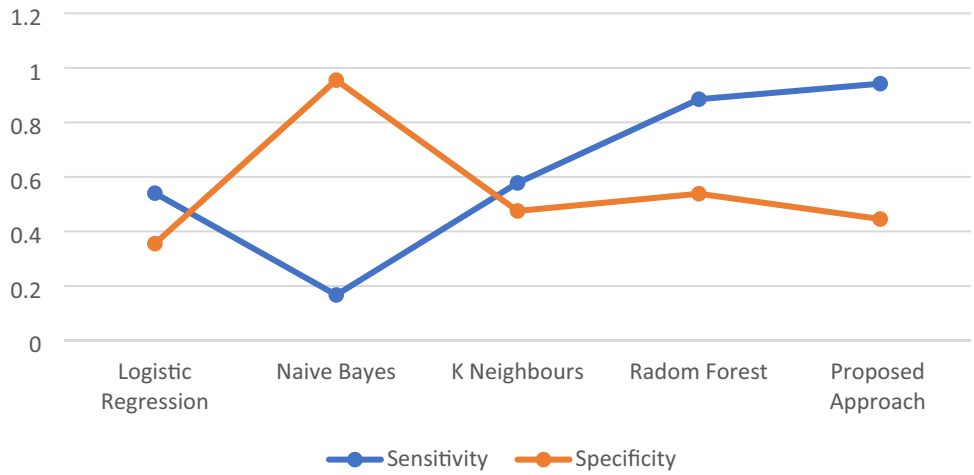


Fig. 18 Comparative Analysis of Sensitivity and Specificity: Standard Algorithms vs. Proposed Approach

True Negative values, we see that the highest number of True Positive value stands with Random Forest while the highest number of True Negative values stand with Naïve Bayes but it has a low accuracy rate compared to Random forest that is 55.83% compared to 72.76%. This is because Naïve Bayes' has the lowest True Positive rate compared to others, while random forest has the highest True Positive and second most in True negative.

For Naïve Bayes, True Negative values are cancelled out by its high rate of False Negative since it generates the highest rate in both True Negative and False Negative.

Hence, the best that could be achieved from the graphs above is Random Forest with its favourably less False Negative value, high True Positive value and high True Negative value. Even though Random Forest's False Positive value is high, it cannot cancel out the high True Positive value of 431 compared to 187. Taking Random Forest and treating and optimizing it gave the True Positive value of 650 as shown in Figs. 15 and 16 with an accuracy of 82.96% as mentioned earlier.

The performance measures Sensitivity, Specificity, Precision, Negative Predictive Value, False Positive Rate, False Discovery Rate, False Negative Rate, Accuracy, F1 Score, and Matthews Correlation Coefficient were additionally considered to compare the proposed approach with other classifiers as summarized in Table 4.

Figure 18 presents a comparative analysis of Sensitivity and Specificity for the standard algorithms (Logistic Regression, Naive Bayes, K Neighbours, Random Forest) and our proposed approach. The proposed approach demonstrates the highest Sensitivity of 0.942, surpassing the other classifiers. Additionally, the Naive Bayes classifier exhibits the highest Specificity value of 0.9549. However, the proposed approach and other classifiers show relatively lower Specificity values ranging from 0.4455 to 0.5383.

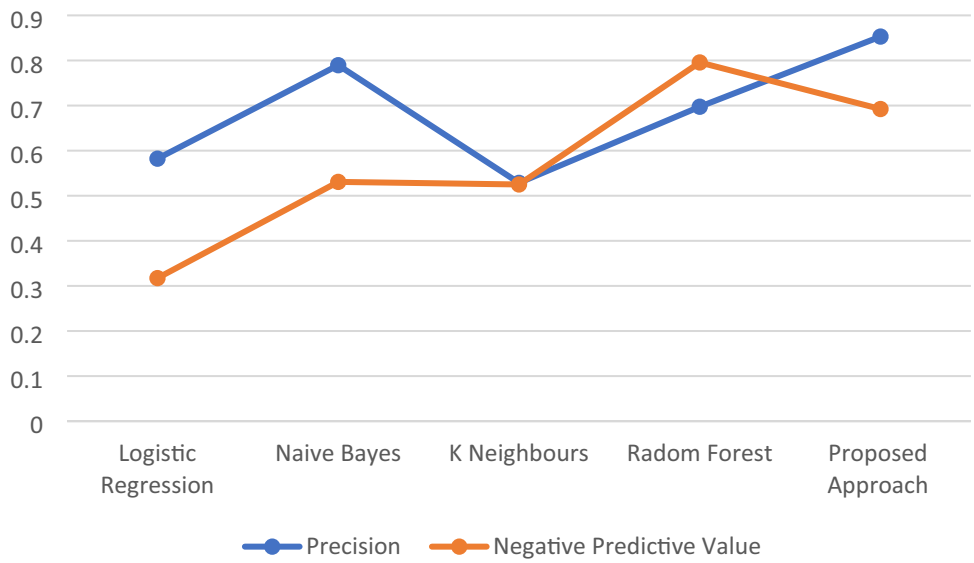


Fig. 19 Comparative Analysis of Precision and Negative Predictive Value: Standard Algorithms vs. Proposed Approach

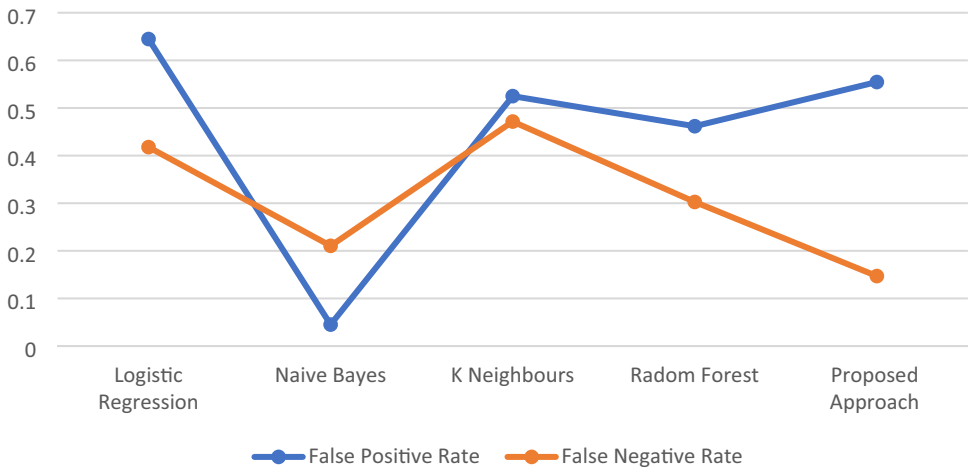


Fig. 20 Comparative Analysis of False Positive Rate and False Negative Rate: Standard Algorithms vs. Proposed Approach

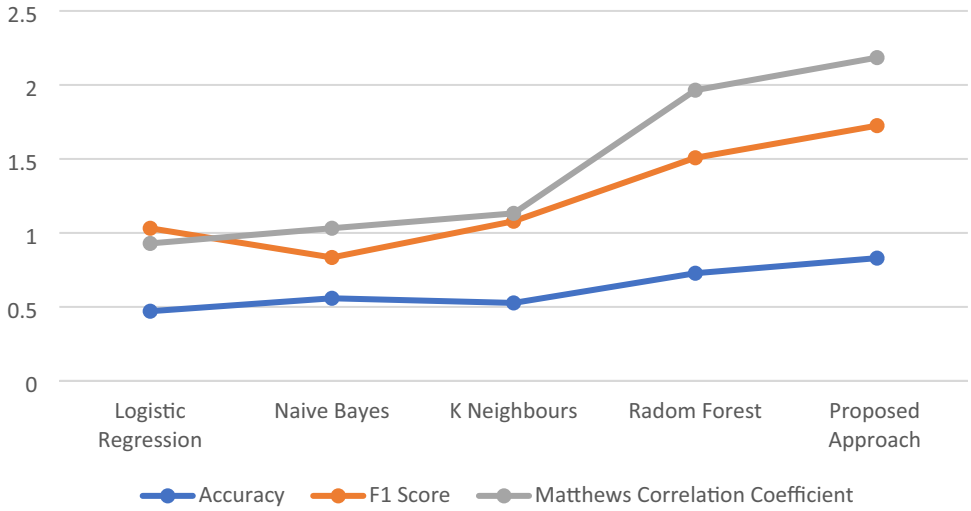


Fig. 21 Comparative Analysis of Accuracy, F1 Score and MCC: Standard Algorithms vs. Proposed Approach

Figure 19 illustrates a comparative analysis of Precision and Negative Predictive Value for the standard algorithms and our proposed approach. The proposed approach achieves the highest Precision value of 0.853, indicating its ability to minimize false positive predictions. The Naive Bayes classifier also exhibits a respectable Precision value of 0.7895. Moreover, the Random Forest classifier achieves the highest Negative Predictive Value of 0.7956, suggesting its proficiency in accurately identifying non-fault-prone instances.

Figure 20 displays a comparative analysis of False Positive Rate (FPR) and False Negative Rate (FNR) for the standard algorithms and our proposed approach. The proposed approach exhibits a higher FPR of 0.5545 compared to the other classifiers. Conversely, the Naive Bayes classifier demonstrates the lowest FPR of 0.0451. Regarding FNR, the proposed approach achieves the lowest value of 0.058, indicating its ability to minimize the misclassification of fault-prone instances. The Naive Bayes classifier, however, exhibits the highest FNR of 0.833.

Figure 21 provides a comparative analysis of Accuracy, F1 Score, and Matthews Correlation Coefficient (MCC) for the standard algorithms and our proposed approach. The proposed approach demonstrates the highest Accuracy of 0.8296, outperforming all the other classifiers. It also achieves the highest F1 Score of 0.8953, indicating its ability to achieve a good trade-off between precision and recall. Moreover, the proposed approach and the Random Forest classifier achieve relatively higher MCC values of 0.4597 and 0.4568, respectively. The Naive Bayes classifier demonstrates the highest MCC value of 0.1976, while the Logistic Regression classifier exhibits a negative MCC value. The detailed performance measures for all the classifiers are summarized in Table 4.

In summary, the analysis of the performance measures, as illustrated in Figs. 18, 19, 20, and 21, and summarized in Table 4, reveals that our proposed approach consistently outperforms the other classifiers across various metrics. It exhibits higher Sensitivity, Precision, Negative Predictive Value, Accuracy, F1 Score, and MCC compared to the

alternatives. Additionally, the proposed approach achieves a good trade-off between precision and recall, as demonstrated by the highest F1 Score. These results strongly suggest that our proposed approach is effective and robust in predicting software faults using the NASA jm1 dataset. We have planned to use different datasets to check the accuracy of our proposed approach in future.

Conclusion

Based on the analysis, we understand that the results provided by the Random Forest technique, specifically the proposed approach, demonstrate promising potential for predicting software defects using the NASA JM1 dataset. The improved Random Forest algorithm achieves higher accuracy and precision compared to the standard algorithms, indicating its effectiveness in identifying and preventing software defects before they occur. The comparative analysis of performance measures, as discussed in the result analysis section, supports the superiority of our proposed approach.

Specifically, our proposed approach achieves a sensitivity of 0.942, demonstrating its ability to accurately identify positive instances and predict software faults. The precision of 0.853 further emphasizes the model's ability to minimize false positive predictions. Moreover, the proposed approach achieves an accuracy of 0.8296, surpassing all other classifiers, and an F1 Score of 0.8953, indicating a good trade-off between precision and recall.

The feature importance analysis conducted on the Random Forest model provides insights into the variables that are most relevant for predicting software defects. This information can help developers prioritize their efforts and allocate resources more effectively, enhancing the overall quality and reliability of software systems.

Furthermore, the properties of Random Forest, such as its scalability, interpretability, and non-parametric nature, make it a valuable tool in machine learning applications. Its high accuracy and feature importance analysis contribute to its usefulness in various industries, including finance, healthcare, marketing, and more. For example, Random Forest can be applied to recommender systems, fraud detection, natural language processing, and autonomous vehicles, among others, providing valuable insights and improving decision-making processes.

Hyperparameter tuning plays a crucial role in optimizing the Random Forest model's performance. By carefully selecting hyperparameter values such as the number of trees, tree depth, and minimum sample leaf size, we can further improve the accuracy and precision of the model. Techniques such as grid search, random search, and Bayesian optimization are commonly used for hyperparameter tuning. Each technique has its advantages and disadvantages, and careful consideration should be given to choose the most suitable approach based on the specific requirements and constraints of the application.

In summary, our proposed approach for software fault prediction using the NASA JM1 dataset, based on an improved Random Forest algorithm, outperforms the standard algorithms in terms of accuracy and precision. The numerical results from the comparative analysis support the effectiveness of our approach. As machine learning and data science continue to evolve, similar approaches are expected to gain wider adoption in software development and other fields. The importance of continued research in this area is evident, as it can lead to further advancements in predictive modeling, classification, feature selection, and overall software quality improvement.

In envisioning future avenues of research, our work lays the groundwork for several promising directions. Further refinement of the proposed approach could involve exploring advanced ensemble techniques beyond Random Forest to discern their impact on software fault prediction accuracy. Additionally, investigating the integration of explainable AI methods could enhance the interpretability of predictive models, especially in safety–critical applications. Furthermore, the exploration of alternative datasets and the extension of the study to diverse software development environments could contribute to a more nuanced understanding of the proposed methodology's generalizability. Finally, delving into the dynamic nature of software systems and the incorporation of temporal aspects in predictive modeling represent exciting prospects for future research endeavors in the domain of software fault prediction.

Acknowledgements This work was supported by MIT Manipal, Manipal Academy of Higher Education, Manipal, India, under the Student Exchange Program MESVCC 2022 at Manipal Institute of Technology. We would like to thank the Head of the Department of ICT for providing resources and infrastructure for this work.

Funding Open access funding provided by Manipal Academy of Higher Education, Manipal. The authors received no specific funding for this study.

Data Availability The data that support the findings of this study are openly available in Promise Repository at http://promise.site.uottawa.ca/SERepository/datasets/jm1.arff.

Declarations

Conflict of Interest The authors declare that they have no conflicts of interest to report regarding the work.

References

1. Burnstein S. Practical software testing: a process-oriented approach. Springer Science & Business Media; 2006.
2. Wallace DR, Kuhn DR. Failure modes in medical device software: an analysis of 15 years of recall data. Int J Reliab Qual Saf Eng. 2001;8(04):351–71.
3. Jones C. The year 2000 software problem: Quantifying the costs and assessing the consequences. ACM Press/Addison-Wesley Publishing Co.; 1997.
4. Kuehn BM. Pacemaker recall highlights security concerns for implantable devices. 2018; pp. 1597–1598.
5. Kaliraj S, Chandru N, Wahi A. A reliability framework of component based software system using Kal-Chan path selection algorithm. Int Rev Comput Softw. 2013;8(2):605–12.
6. Kaliraj S, Bharathi A. Path testing based reliability analysis framework of component based software system. Measurement. 2019;144:20–32.
7. Fenton NE, Neil M. A critique of software defect prediction models. IEEE Trans Softw Eng. 1999;25(5):675–89.
8. Alzubaidi L, Bai J, Al-Sabaawi A, et al. A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications. J Big Data. 2023;10:46.
9. Ghojogh B, Crowley M. The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial. arXiv preprint 2019; arXiv:1905.12787.
10. Thaher T, Khamayseh F. A classification model for software bug prediction based on ensemble deep learning approach boosted with SMOTE Technique. In: Congress on Intelligent Systems, 2020; pp. 99–113.
11. Khan F, Kanwal S, Alamri S, Mumtaz B. Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction. IEEE Access. 2020;8:20954–64.
12. Liu Y, Wang Y, Zhang J. New machine learning algorithm: Random forest. In: Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14–16, 2012. Proceedings 3, 2012; pp. 246–252.
13. Shin Y, Williams L. An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical software engineering and measurement, 2008; pp. 315–317.
14. Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Softw Eng. 2010;37(6):772–87.
15. Hammouri A, Hammad M, Alnabhan M, Alsarayrah F. Software bug prediction using machine learning approach. Int J Adv Comput Sci Appl. 2018;9(2).
16. Moustafa S, ElNainay MY, El Makky N, Abougabal MS. Software bug prediction using weighted majority voting techniques. Alex Eng J. 2018;57(4):2763–74.
17. Porter AA, Selby RW. Empirically guided software development using metric-based classification trees. IEEE Softw. 1990;7(2):46–54.
18. Pascarella L, Palomba F, Bacchelli A. Fine-grained just-in-time defect prediction. J Syst Softw. 2019;150:22–36.
19. Goel L, Sharma M, Khatri SK, Damodaran D. Implementation of data sampling in class imbalance learning for cross project defect prediction: an empirical study. In: 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR), Kitakyushu, Japan, 2018, pp. 481–485. https://doi.org/10.1109/ICIEV.2018.8641006
20. Ganguly KK, Hossain BM. Evaluating the effectiveness of conventional machine learning techniques for defect prediction: a comparative study. In: 2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR), 2018; pp. 481–485.
21. Singh P, Pal NR, Verma S, Vyas OP. Fuzzy rule-based approach for software fault prediction. IEEE Trans Syst Man Cybern Syst. 2016;47(5):826–37.
22. Probst P, Wright MN, Boulesteix A-L. Hyperparameters and tuning strategies for random forest. Wiley Interdiscipl Rev Data Min Knowl Discov. 2019;9(3): e1301.