Journal of Electronic Testing (2023) 39:631–657
https://doi.org/10.1007/s10836-023-06089-0

Effective Software Mutation-Test Using Program Instructions Classification

Zeinab Asghari[1] · **Bahman Arasteh**[2,3] · **Abbas Koochari**[1]

Received: 20 June 2023 / Accepted: 12 October 2023 / Published online: 9 January 2024

Abstract
The quantity of bugs that a software test-data finds determines its effectiveness. A useful technique for assessing the efficacy of a test set is mutation testing. The primary issues with the mutation test are cost and time requirements. Close to 40% of the injected bugs in the mutation test are effect-less (equivalent). Reducing the number of generated total mutants by decreasing equivalent mutants and reducing the execution time of the mutation test are the main objectives of this study. An error-propagation aware mutation test approach has been suggested in this research. Three steps make up the process. To find a collection of instruction-level characteristics effective on the error propagation rate, the data and instructions of the input program were evaluated in the first step. Utilizing supervised machine learning techniques, an instruction classifier was developed using the prepared dataset in the second step. After classifying the program instructions automatically by the created classifier, the mutation test is performed only on the identified error-propagating instructions; the identified non-error-propagating instructions are avoided to mutate in the proposed mutation testing. The conducted experiments on the set of standard benchmark programs indicate that the proposed method causes about 19% reduction in the number of generated mutants. Furthermore, the proposed method causes a 32.24% reduction in the live mutants. It should be noted that the proposed method eliminated only the affectless mutants. The key technical benefit of the suggested solution is that mutation of the instructions that don't propagate errors is avoided. These findings can lead to a performance improvement in the existing mutation-test methods and tools.

Keywords Software test · Equivalent mutant · Error propagation · Machine learning · Instruction classification

1 Introduction

Nowadays, software is regarded as an important part of modern human life. In fact, software failures in different applications may have irreversible financial and humanistic impacts [5, 6]. Hence, by using quality assurance methods, software failure can be prevented and avoided. Software reliability (as one of the quality metrics) is impacted by the number of bugs in the software [3]. Software reliability may be remarkably enhanced by detecting and removing bugs in each stage of software development [2]. Software testing is a primary technique for evaluating and assuring the reliability of a software product. Software bugs made by a developer may lead to errors and finally failure during program execution. It should be noted that the cost of software testing accounts for 50% of the total software development cost [21]. The effectiveness of a test depends on the number of detected bugs by it. Ideally, close to 100% of the bugs in a program can be detected by effective test data.

Mutation testing is a practical method for evaluating the effectiveness of a set of test data (test set). The impetus behind mutation testing is to inject bugs, namely mutants, into program source code. In this field, a program with an injected bug (mutation) is named mutant. Such modifications (injected bugs) are realized and implemented by means of a series of mutation operators. Consequently, the new buggy-programs (mutants) are generated during mutation testing. Indeed, a mutant simulates the behavior of a buggy

Responsible Editor: Y. K. Malaiya.

✉ Bahman Arasteh
bahman.arasteh@istinye.edu.tr

Zeinab Asghari
zeinab.asghari@srbiau.ac.ir

Abbas Koochari
koochari@srbiau.ac.ir

[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran

[2] Department of Computer Engineering, Tabriz Branch, Islamic Azad University, Tabriz, Iran

[3] Department of Software Engineering, Faculty of Engineering and Natural Science, Istinye University, Istanbul, Turkey

program. The main objective of mutation testing is to evaluate the capability of a test set for detecting mutations (injected bugs). In Table 1, three mutant programs are created from the main program. Line 5 of the original program has changed with three mutations. To enhance the efficacy of mutation testing, a large number of mutants should be generated systematically. The number of generated mutants depends on the number of lines of the source code. In programs with a high number of code lines, a large number of mutants are produced. Hence, cost and time consumption are two of the main problems with the mutation test; this is regarded as one of the most serious challenges in the realm of software testing.

According to similar research [25], 40% of the injected bugs in the mutation test have no impact (equivalent). As a result, just a portion of the bugs that were injected are active and causing output in the application. The existence of effect-less mutations causes the mutation test to be less effective, take longer, and cost more money. If a mutant program differs syntactically from the original program but has the same outcomes, the mutant program is equivalent. The behavior of the equivalent mutants is the same as that of the original program. One of the major challenging problems is equivalent mutants' identification, because identifying the equivalent mutants is an undecidable problem. The following are the paper's goals:

- Reducing the number of generated mutants by decreasing equivalent mutants.
- Reducing the execution time of mutation testing by decreasing the number of generated mutants.
- Reducing the cost of mutation testing by decreasing the number of generated mutants.
- Reducing the number of generated equivalent mutants generated.

In this paper, program data and instructions were analyzed and the set of syntactic features effective in the error-propagation rate of program instructions were identified. Then, program instructions are classified according to the identified features using supervised machine-learning algorithms. In fact, by capitalizing on machine learning algorithms, program instructions are ranked according to error propagation. Finally, mutation operators are applied only to the instructions with a higher error propagation rate.

The contributions of the present study are as follows:

- Extracting the static and dynamic features effective on error-propagation at the level of program instructions.
- Producing a dataset (as training-data set) that includes the instruction-level features for evaluating the instructions' error-propagation rate.
- Generating an instruction classifier to classify the program instructions using machine learning algorithms and the produced training-data set.
- Applying the mutation operators only on the most error-propagating instructions identified by the classifier and eliminating the non-error-propagating instructions in the mutation test.
- Identifying the equivalent mutants using the instruction classifier and eliminating them from the mutant set.

The seven sections of the paper are as follows: The second section provides a brief review of important mutant reduction techniques. Section 3 illustrates Supervised machine learning algorithms used in article. Section 4 explains the proposed method. The environment and tools that are utilized to evaluate the proposed approach are described in the first half of Section 4. Datasets and evaluation criteria are also covered in this section. In the second part of Section 4, the study's findings are provided and analyzed. Section 5 presented a review of mutation test environments. Section 6 is a complete description of the evaluation of the proposed system. Eventually in Section 7, the study's conclusion as well as future research directions are provided.

Table 1 Original program and the generated mutants

Original program	Mutant 1

1. class Class A{
2. public intinc(int a, int b){
3. for(b<10){
4. a++;
5. b=b+2;
6. }
7. return a;
8. } }
	1. Class Class A {
	2. public intinc(int a, int b){
	3. for(b<10){
	4. a++;
	5. b=b-2;
	6. }
	7. return a;
	8. } }

Mutant 2	Mutant 3

1. class Class A{
2. public intinc(int a, int b){
3. for(b<10){
4. a++;
5. b=b*2;
6. }
7. return a;
8. } }
	1. class Class A{
	2. public intinc(int a, int b){
	3. for(b<10){
	4. a++;
	5. b=b/2;
	6. }
	7. return a;
	8. } }

2 Related Works

Mutation testing is applied for evaluating the quality of test data. The rationale behind the mutation test is that a series of bugs are used so as to simulate real programing bugs. Such buggy programs (mutants) are produced as a result of making minor syntactic changes in the program source code. Equation 1 states that the percentage of discovered mutants (killed mutants) in test data determines a test's score [31]. All of the inserted bugs are found and eliminated using efficient test data. Live mutants are mutations that give the same test data results as the original program. The test set is deemed sufficient, and the test procedure is terminated if the mutation score of the test set is equal to 100 percent. The mutation score is influenced by the presence of equivalent mutants. The output of the comparable mutants is identical to that of the original program. In one of the articles with the aim of reducing the number of mutations produced, the combination of clustering and sorting methods is used to reduce the number of mutations produced [54]. Another method has also been proposed in which the combination of clustering and sorting methods is used to reduce the number of mutations produced. Another method has also been proposed in which have been sorted mutant branches based on their dominance degrees [15, 16].

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{All Mutants} - \text{Equivalent Mutants}} \quad (1)$$

Researchers have proposed many strategies for lowering the cost of mutation testing. Here's a review of several key techniques:

Mutant Sampling It is regarded as one of the simplest methods for reducing the number of mutations [12]. Mutant sampling involves performing mutations on a limited sample of the produced mutants. Numerous studies have examined the percentages of various samples, ranging from 10 to 40% [52]. The effect of the 10% sample percentage was only 16% smaller than the entire set of generated mutants, according to the experimental results. As a result, techniques for assessing mutations with a 10% sampling percentage can still be a good option. This is consistent with King's research [27]. Through tests, Papadakis and Malevris [42] investigated the effectiveness of several mutation sampling techniques (from 10 to 60% in 10% steps).

Mutant Selection Selective mutation is a rough method for lowering the number of mutants. In order to decrease the number of executive mutations, authors suggested random selection mutation. Randomly, just a tiny portion of the mutations are investigated [1]. A method known as limited mutation only considers a limited number of mutants while disregarding the majority of them. The method selects just some of the mutation operators to make mutants [39]. To expand on this strategy, the authors examined the efficacy of several mutation operator sets [38]. The results show that just five operators out of a total of 22 are needed to measure the success of mutation testing. The authors suggested six operators for estimating the number of mutation operators that are suitable [8]. A set of 10 operators was developed by combining these operators, which eliminated 65% of the mutations while maintaining the test validity. Other research examined the effectiveness of using just one or two mutation operators. In comparison to the dependent mutation operator, Authors assessed the effectiveness of utilizing mutation with one or two assignment mutation operators [52]. According to the experiment results, the number of matching mutations can be decreased by up to 67%, while only 5% of the test efficacy is lost. In addition, multiple studies have demonstrated that inserting these mutations has no effect on the quality of the test cases generated. Researchers looked at the differences between sampling mutation and selective mutation [57]. Two sample methodologies were compared to three selection procedures. The selective mutation was shown to be less effective than the sampling mutation. Finally selecting and sampling mutations be used in combination to generate promising results [56]. Another method has been proposed by researchers, in which certain pathways are produced based on unique characteristics, and based on this, the production of mutations can be somehow controlled This algorithm is an optimization algorithm suitable for identifying optimal paths with priority [14].

Minimum Mutant Sets The results of experiments indicate that by concentrating on a minimum set of mutants, a considerable proportion of mutants might be eliminated [32]. Researchers have made an effort to determine the fewest mutations necessary to fully cover their set, which would be adequate for determining the minimum set's ability. The experiment that used the fewest changes to program source code was originally proposed by another authors [28]. The gathered data indicates that even for mutations that are scarcely destroyed, just a modest portion of the produced mutations (9%) is required to cover the whole set. Other researchers investigated this issue both theoretically and experimentally [29]. Dynamic sharing was used to lower the number of mutations. Given a test set, the x mutation is dynamically translated into the y mutation; test cases that kill x also kills y. When the dynamic subset was examined using the C programming language, it was found that just 12% of the generated mutations were necessary to cover the whole set. Last but not least, researchers investigated if establishing the association between frequent mutations might be done using dynamic and static analytic approaches [29, 30]. They found that for better results, static and dynamic analysis

techniques should be used. Another innovative method proposes an optimized decentralized adaptation logic for modeling Self Adaptive Software Systems (SAS) which exploits the multi-agent concept. Each subsystem has an objective and uses an Artificial Bee Colony metaheuristic to achieve local optimization which in turn leads to the optimization of the whole distributed system [9].

Strong, Weak and Hard Mutations Researchers have developed a variety of strategies for reducing the cost of the mutation test's application, in addition to limiting the number of generated mutants to reduce mutation costs. The weak mutation strategy is one of Howden's strategies [25]. By omitting the whole implementation of the main program and its mutations, weak mutation aims to lower the processing cost associated with mutation avoidance. To achieve this, the weak mutation lays forth the requirements that a mutation must meet in order to be labeled as a dead mutation. To compare the final output of the main program and the modified program, the internal states of the programs are compared immediately after applying the mutation or altered components. It should be noted that when compared to a "weak mutation," the average mutation is referred to as a "strong mutation. Researchers proposed the strong mutation as a middle ground between strong and weak mutations [53]. They claimed that the internal states of the main program and its mutations at any point could be compared. Weak mutations are useful in many investigations. Researchers built a structure for the FORTRAN77 software and then assessed its effectiveness and usefulness [37]. The results showed that weak mutations reduced manual effectiveness because fewer related mutations were evaluated. In this article, the authors used a range of techniques to examine the efficacy of weak mutations. They concluded that weak mutation is more cost-effective. Researchers proposed novel mutation test execution methods known as higher rank (order) mutation [25, 28, 53]. First rank (order) mutations and higher order mutations are the two categories used in this technique to classify mutations. First order mutation is produced if mutational operators are only applied once to the program. Higher level mutations will be created if they are applied more than once, though. This method's motivation is the discovery of unique and rare bugs. Table 2 provides an illustration of a lower-order, higher-order, and first-order mutation. Table 3 displays the key characteristics, advantages and disadvantages of some of the strategies described earlier.

3 Supervised Machine Learning Algorithms

Machine learning uses programmed algorithms that learn and optimize their operations by analyzing input data to make predictions within an acceptable range. With the feeding of new data, these algorithms tend to make more accurate predictions. Although there are some variations of how to group machine learning algorithms, they can be divided into three broad categories according to their purposes and the way the underlying machine is being taught. These three categories are: supervised, unsupervised and semi-supervised.

Supervised machine learning algorithms generate a function that map inputs to desired outputs [41]. Supervised learning is fairly common in classification problems because

Table 2 First, second and higher order mutations

Original program p

```
result = 0;
for (int i = 1; i <= 10; i++){
  result = result + i;       }
```

FOM_1

```
result = 0; for (int i = 1; i>= 10; i++) {
  result = result + i;                    }
```

FOM_2

```
result = 0;
for (int i = 1; i <= 10; i++) {
  result = result / i;        }
```

SOM

```
result = 0;
for (int i = 1; i>= 10; i++) {
   result = result / i;       }
```

HOM

```
result = 0;
for (int i = 1; i>= 10; i--) {
   result = result / i;       }
```

Table 3 The related works which proposed to reduce the number of mutants

The methods	Procedure	Merits	Demerits
Mutation sampling:			
[1, 12, 27, 47, 51, 52]	The produced mutations are picked as a subset.	The straightforwardness of the test execution	lower test effectiveness
Selective mutation, limited mutation:			
[8, 28, 35, 38, 39, 42, 52, 56, 57]	choosing a limited number of mutation operators	lowering the number of mutations by 65%	The necessity to combine this approach with mutation sampling due to its poor performance when used alone
Minimum mutation sets:			
[18, 22, 28–30, 32]	Eliminating the covering mutants s.	To cover the whole set, only a tiny portion of the created mutations are needed.	Imprecise
Strong, weak and hard mutations: [24, 25, 28, 53]	Weak mutation: By bypassing the full execution of the program, it lowers the number of mutations Strong mutation: it reduces the number of mutations by contrasting the output of the original program with the output of the modified program. A mixture of strong and weak mutation is referred to as "hard mutation".	They cost less and need less computational resources.	If the entire application is not run, they require comparison and might be inaccurate.

the goal is often to get the computer to learn a classification system that we have created [48]. Supervised machine learning is the construction of algorithms that are able to produce general patterns and hypotheses by using externally supplied instances to predict the fate of future instances. Supervised machine learning classification algorithms aim at categorizing data from prior information. Classification is carried out very frequently in data science problems. Various successful techniques have been proposed to solve such problems viz. Rule-based techniques, Logic-based techniques, Instance-based techniques, stochastic techniques [49].

Decision Tree Decision tree (DT) is one of the earliest and prominent machine learning algorithms. A decision tree models the decision logics i.e., tests and corresponds outcomes for classifying data items into a tree-like structure. One of the algorithms used in this paper is DT. Because Decision trees does not require scaling of data as well. Decision trees were chosen for this case study given their ability to convert datasets into easy-to-understand and yet information-rich graphical displays. The nodes of a DT tree normally have multiple levels where the first or top-most node is called the root node. All internal nodes (i.e., nodes having at least one child) represent tests on input variables or attributes. Depending on the test outcome, the classification algorithm branches towards the appropriate child node where the process of test and branching repeats until it reaches the leaf node [43]. The leaf or terminal nodes correspond to the decision outcomes. DTs have been found easy to interpret and quick to learn and are a common component to many medical diagnostic protocols [13].

Random Forest A random forest (RF) is an ensemble classifier and consisting of many DTs similar to the way a forest is a collection of many trees [11]. DTs that are grown very deep often cause overfitting of the training data, resulting a high variation in classification outcome for a small change in the input data. They are very sensitive to their training data, which makes them error-prone to the test dataset. The different DTs of an RF are trained using the different parts of the training dataset. To classify a new sample, the input vector of that sample is required to pass down with each DT of the forest. Each DT then considers a different part of that input vector and gives a classification outcome. The forest then chooses the classification of having the most 'votes' (for discrete classification outcome) or the average of all trees in the forest (for numeric classification outcome). Since the RF algorithm considers the outcomes from many different DTs, it can reduce the variance resulted from the consideration of a single DT for the same dataset Fig. 1 shows an illustration of the RF algorithm. Another machine learning algorithm used in this paper is RF. Because It is flexible to

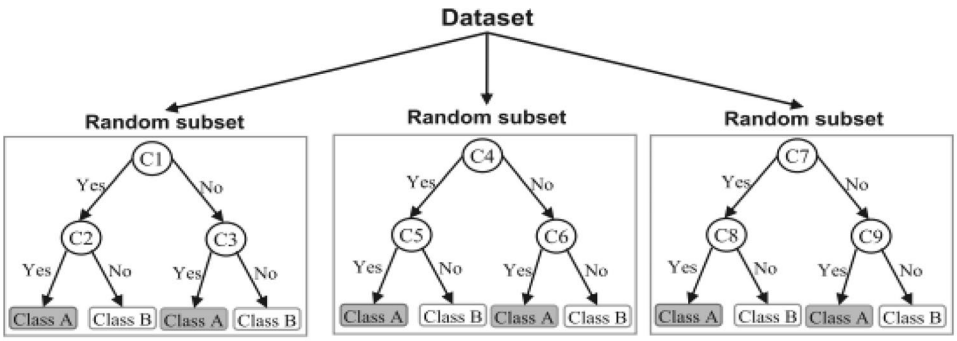

Fig. 1 A brief illustration of a Random Forest

both classification and regression problems and automates missing values present in the dataset.

Artificial Neural Network Artificial neural networks (ANNs) are a set of machine learning algorithms which are inspired by the functioning of the neural networks of human brain. They were first proposed by McCulloch and Pitts [33] and later popularized by the works of Rumelhart et al [44]. In the biological brain, neurons are connected to each other through multiple axon junctions forming a graph like architecture. These interconnections can be rewired (e.g., through neuroplasticity) that helps to adapt, process and store information. Likewise, ANN algorithms can be represented as an interconnected group of nodes. The output of one node goes as input to another node for subsequent processing according to the interconnection. Nodes are normally grouped into a matrix called layer depending on the transformation they perform. Apart from the input and output layer, there can be one or more hidden layers in an ANN framework. Nodes and edges have weights that enable to adjust signal strengths of communication which can be amplified or weakened through repeated training. Based on the training and subsequent adaption of the matrices, node and edge weights, ANNs can make a prediction for the test data. Figure 2 shows an illustration of an ANN (with two hidden layers) with its interconnected group of nodes. This algorithm is also used in this paper. It is non-linear in nature. This allows to model complex relationships and patterns in dataset. It can extract features from dataset. This eliminates manual feature editing. Also, the high speed and parallel processing capability of this algorithm has made it to be used in this paper.

K-nearest Neighbor The K-nearest neighbor (KNN) algorithm is one of the simplest and earliest classification algorithms [10]. It can be thought a simpler version of an NB classifier. Unlike the NB technique, the KNN algorithm does not require to consider probability values. The 'K' is the KNN algorithm is the number of nearest neighbor's considered to take 'vote' from. The selection of different values for 'K' can generate different classification results for the same sample object. Figure 3 shows an illustration of how the KNN works to classify a new object. For $K = 3$, the new object (star) is classified as 'black'; however, it has been classified as 'red' when $K = 5$.

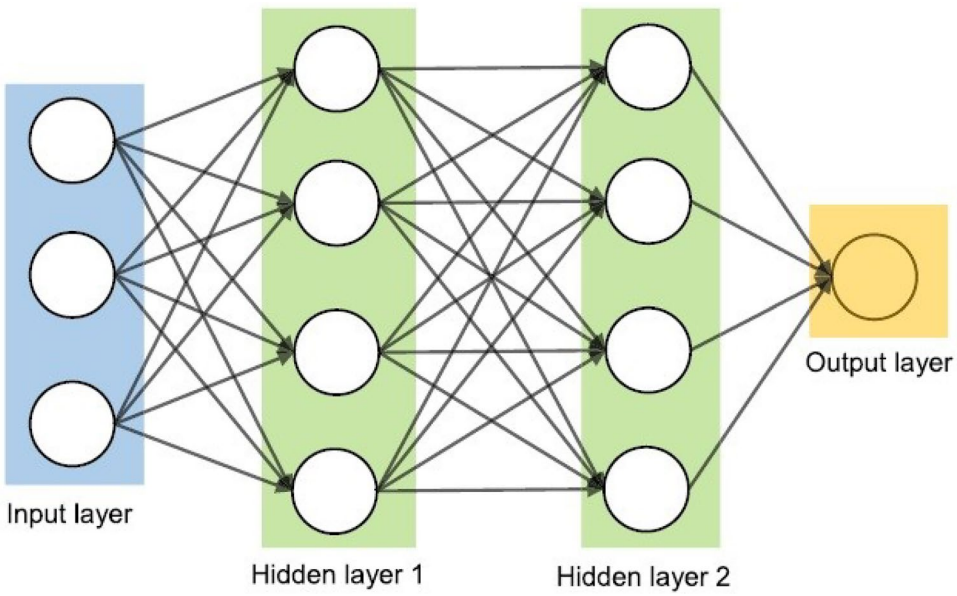

Fig. 2 An illustration of the artificial neural network structure with two hidden layers

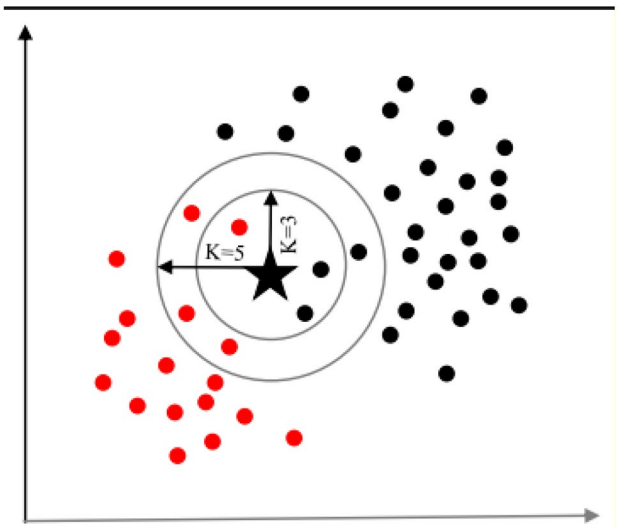

Fig. 3 A simplified illustration of the K-nearest neighbor algorithm

Naïve Bayes Naïve Bayes (NB) is a classification technique based on the Bayes' theorem [23]. This theorem can describe the probability of an event based on the prior knowledge of conditions related to that event. This classifier assumes that a particular feature in a class is not directly related to any other feature although features for that class could have interdependence among themselves [36]. By considering the task of classifying a new object (white circle) to either 'green' class or 'red' class, Fig. 4 provides an illustration about how the NB technique works.

Gradient Boosting Gradient boostingwas first proposed in [19]. GBT models build ensembles of decision trees and apply the boosting principle to learn a tree structure where each new tree is built to approximate the negative gradient of the empirical loss function in order to correct the errors made by previous trees in the ensemble. These trees are typically weak learners, i.e., the size of the individual trees is typically kept small. Furthermore, in each iteration of the training phase, only a random subsample of the data instances (rows) and features (columns) are used, in order to prevent overfitting [20]. The final prediction is calculated by combining the individual predictions with coefficients learned during the training phase. This is in contrast to random forests (RF), another tree-based ensemble model, where full-grown trees are built, and their predictions are combined uniformly. One reason as to why we would consider using GBT algorithm in this paper is this is generally more accurate compared to other models. Another advantage is that this algorithm trains faster and handles missing values natively.

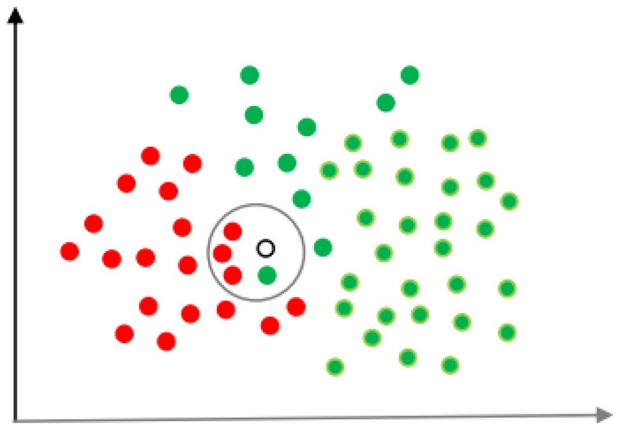

Fig. 4 An illustration of the Naïve Bayes algorithm

4 The Proposed Method

The primary goal of this study is to improve the efficacy of mutation testing. Mutation testing is thought to have a number of issues, one of which being the existence of equivalent mutants. The quantity and kind of program instructions affect how many equivalent mutants there are. Decreasing the number of equivalent mutations increases the value of the mutation score. Figure 5 presents an overview of the suggested method's stages. The proposed method is divided into two main parts: front-end and back-end of proposed method. Steps 1 to 3 are located in the front part and steps 4 and 5 are located in the back part. Finally, the output of this method will include step 6. The output of the front part includes the dataset required for machine learning training and creating instruction classifiers. The output of the back part is the classifier created from the instructions.

In this paper, program data and instructions were analyzed and the syntactic features effective on error propagation at the level of program instructions were identified. Syntactic features include static and dynamic features. Static features are extracted statically without program execution. However, the identification of dynamic features requires test cases and program execution. Then, program instructions are classified according to their error propagation rate. In this paper, supervised machine learning algorithms were used to classify the program instructions. In fact, by capitalizing on different machine learning algorithms (*Gradient Boosted Trees, Decision Tree, Deep learning and Random Forest*), program instructions are ranked according to error propagation. Finally, mutation operators are applied only to the instructions with a higher error-propagation probability.

4.1 Program Analysis

The proposed method classifies a program's instructions based on the error-propagation rate of the instruction.

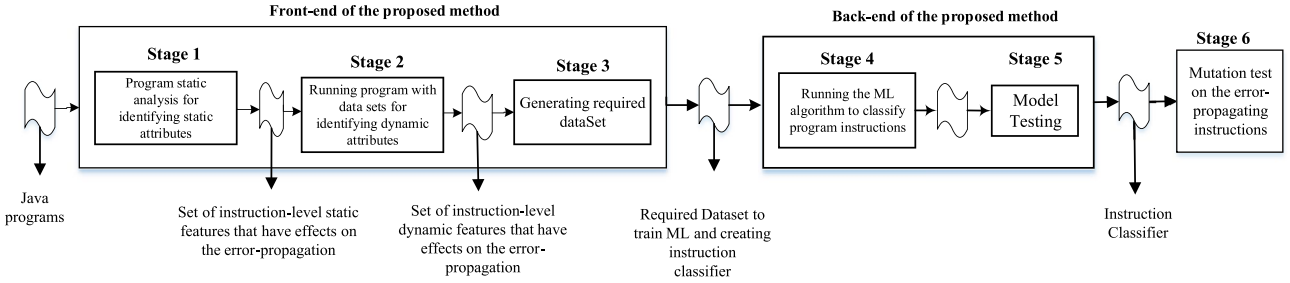

Fig. 5 Stages of the proposed method

The error-propagation rate of an instruction in a program depends on some unknown features. Identifying these effective features is a contribution of this study. The proposed classifier was created using a specific dataset. The required dataset includes the specific features of the instructions of the selected benchmark programs. Firstly, the effective instruction-level features are identified by analyzing the source code of the selected benchmark programs. Indeed, the degree of error-propagation rate in each instruction is measured according to the identified features. In this study, the identified features that have an impact on the error propagation rate are shown in Table 4. The purpose of the first stage of the proposed method is to identify those features of the program instructions that are required to create a dataset. The required dataset will be prepared in stage 2. This dataset is used to create an instruction-classifier by the machine learning algorithm at stage 3. The required dataset includes some strategic information (features) about the program instructions that have an impact on the error propagation rate. The objective dataset includes static and dynamic features, respectively. The static and dynamic features of the required dataset to create the instruction classifier to be identified. The static features are quantified by the static analysis, and the dynamic features of the program are quantified by the dynamic analysis.

Static Analysis At this stage, each instruction within the selected benchmark programs is statically analyzed to quantify the value of the static feature depicted in Table 3. This step of the proposed method shown in Fig. 5, the static features of the program are extracted from each program instruction. These features are listed in Table 4. To find these features, as it is clear from the names of this, there is no need to run the program, and these characteristics can be quantified using the CFG graph. The CFG graph is depicted in Fig. 7 for the factorial program. These features somehow affect the program error propagation rate and are therefore of interest. All these details are done in the first step of the proposed method shown in Fig. 5. Each row in the dataset is related to an instruction in a dataset. The control flow graph (CFG) of the program and its description in Tables 5 and 6 are used to quantify the static features shown in Table 4 for the benchmark programs'

Table 4 The identified static and dynamic features of the program instructions

	Attribute	Description
Static Features	Number of Variable Definitions	The number of variables defined in an instruction
	Number of Computational operators	The number of mathematical operators applied in an instruction
	Number of Conditional operators	The number of conditional operators used in an instruction
	Number of Variable usages	Number of variables used in the instruction
	Numerical data value	The presence or lack of numerical value in an instruction
	Data dependency	The number of next instructions which has data dependency on the result of the current instruction
	Nesting level	Accessibility of the instruction
	Control dependency	The number of next instructions which has control dependency on the result of the current instruction
	Instruction length	$N1^a + N2^b$
	Instruction complexity	$(N2/n2^c)*(n1^d/2)$
Dynamic Features	Average Runtime	Average Number of Run

[a]Total number of operators in the instruction

[b]Total number of operands in the instruction

[c]Number of unique operands

[d]Number of unique operators

```
public static long fact() {
1.int num, fact_num = 1;
2.scanner num = new scanner(System.in);
3.if (num < 0)
4.  System.out.println ("Factorials
"+num+" defined
    only on non-negative integers.");
5.else
6. for (int i = 1; i <= num; ++i)
7.    fact_num *= i;
8.system.out.println("fact of "+num+" is
"+fact_num);
}
```

Fig. 6 A source code of a factorial program as case study 1

instructions. CFG represents the program's executing paths. CFG is essential for static analysis of the source code based on the selected features. Firstly, the CFG is extracted from the source code and includes nodes and the arcs; the nodes include a block of non-branch instructions, and the arcs represent the execution flows of the program. In general, a basic block (BB) includes a maximum sequence of program instructions (instructions between two branch instructions). The Factorial program is one of the applied benchmark programs in the experiments which were investigated. Figure 6 illustrates the source code of the factorial program, and Fig. 7 shows the generated CFG for the program. The required CFG is generated automatically from a source code by different open-source tools like Visustin *(*https://www.aivosto.com/visustin.html*) in a polynomial time complexity.*

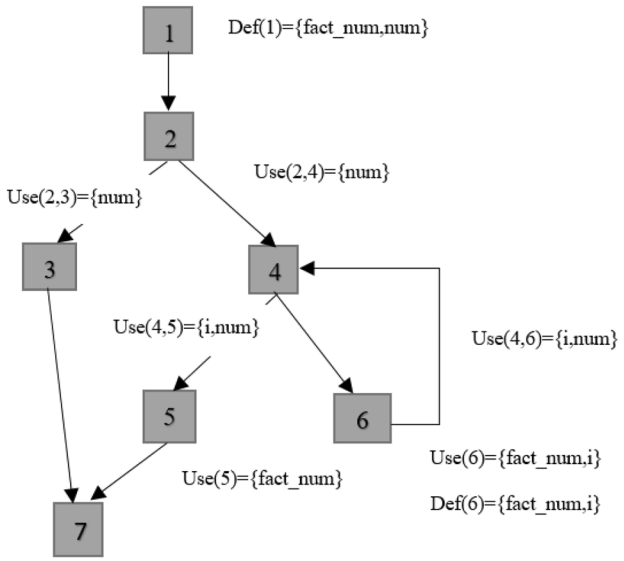

Fig. 7 The CFG of the source code represented by Fig. 2

Table 5 Def-Use (DU) paths of each data in the program shown in Fig. 6

variable	DU-paths
i	(1, (4,5)), (1, (4,6)), (6,6)
fact_num	(1,6), (6,6), (6,5)
num	(1, (2,3)), (1, (2,4)), (1, (4,5)), (1, (4,6))

In a CFG, the nodes with more than one output edge include branch instructions and control data. Table 5 shows the def-use paths (DU paths) for each data in the program using the related CFG. For a given specific variable in the program, all the defined nodes are found. Then, DU paths are extracted among the definition (def) and use (use) nodes. Table 6 gives the related def and use in each node (BB) of the CFG shown in Fig. 7.

The first five features in Table 4 can be set using the source code. The nesting level of an instruction shows the accessibility of the instruction. If the instruction is not in an if instruction, its nesting level is 0; if it is in an if instruction, then its nesting level is 1. The nesting level of an instruction is calculated by the source code analysis. Data and control dependency features are quantified by the generated CFG. The variable used in an instruction determines data dependency of that instruction to another instruction. The control dependency shows the degree of dependence of the instruction on the execution of the parent nodes, which means that if the parent node is not executed, the child node will not be executed either. The number of data and control dependencies of the instruction are obtained from the CFG of the program. The other two features of an instruction are its length and the complexity that can be computed by using the Halstead complexity formula Sommerville [46].

Dynamic Analysis Dynamic analysis is considered to be a method that helps to obtain useful information based on executive paths. At this stage, the dynamic features that are depicted in Table 4, are quantified dynamically for the

Table 6 Definition and use instructions into the nodes of the CFG illustrated in Fig. 7

Node or Edge	Definition (D)	Uses (U)
1(start block)	fact_num,num	
2,3 (edge)		num
3,4 (edge)		num
4,5 (edge)		i,num
4,6 (edge)		i,num
6 (loop block)	fact_num,i	fact_num,i
7 (final block)		

benchmark programs' instructions. It is time to extract the dynamic feature of the instructions. This feature, which is the number of execution of instructions, is obtained from the execution of each program. To get the values of this feature, we need a data set that must be prepared as code coverage. The reason for this is that the code coverage method guarantees that each instruction is executed at least once. For each program, we need 100 executions in the program. Based on these executions, we can estimate the average execution of each instruction. Quantifying these features requires dynamic analysis of the program at run time. Table 7 gives the number of executions of each instruction in a factorial program using a single test data (5). The average number of executions of each instruction is regarded as the dynamic feature which is effective on the error-propagation rate. The program with the test cases should be executed to compute the number of executions of each instruction. Before executing the benchmark program, the required test case should be designed. In this study, edge coverage-based test data is generated for the benchmark programs. Edge coverage covers all the edges in the CFG of the program. Generating test cases to obtain the number of executions of each instruction can be done using the ACO algorithm [45], or the SDA algorithm [14]. In this method, we have manually created test cases based on the edge coverage method and run them on each application.

The test cases that are designed in this way can execute all instructions of the program under test at least once. In order to quantify the number of executions of a program instruction, the program should be executed by the generated test data. Table 7 depicts the average number of executions of the instructions in the factorial programs that have been quantified after real executions by the generated coverage-based test data. Table 8 depicts the quantified features for the instructions of the factorial benchmark program. For example, for factorial source code in Fig. 6, Data Dependency type1 in line seven of the program is 2. Due to the data dependence of the two variables max and x[i] on two instructions 4 and 6, the value of this feature is 2. Data dependency type 2 is

Table 7 The CFG of the source code represented by Fig. 6

Line Number	Avg Num. of Run
1	1
2	1
3	1
4	1
5	1
6	5
7	5
8	1

Table 8 The quantified values of the features for the *factorial* benchmark program

Line Number	AVG. Num. of Run	Num. of Variable Definition	Num. of Computational Operators	Num. of Conditional Operators	Num. of Variable usage	Num. of intermediate Data	Num. of Data Dependency type1	Num. of Data Dependency type2	Num. of Control Dependency	Instruction length	Instruction Complexity	Nesting Level
1	1	1	0	0	0	0	0	0	0	7	2	0
2	1	1	0	0	1	0	0	0	0	4	1	0
3	1	0	0	1	1	1	0	0	0	5	1	0
4	1	0	0	0	1	0	0	0	1	5	1	1
5	1	0	0	1	0	0	1	1	1	2	1	0
6	5	2	1	1	2	1	1	1	0	13	4	1
7	5	1	1	0	2	0	2	0	1	6	2	2
8	1	0	0	0	2	0	1	0	0	9	1	1

a node dependency. For example, line instructions 5 and 6 both have data dependencies of node type. The instruction 5 is dependent on instruction 3. The instruction 6 is dependent on instruction 5. Considering that the control dependency shows the degree of dependence of the instruction on the execution of the parent nodes, instructions 4, 5, and 7 are dependent on the parent node, therefore the value of the control dependency of these three instructions will be equal to 1. The instruction length is calculated as the total number of operators and operands. All the brackets, commas, and terminators are considered operators. For example, in Fig. 6, the instruction length of instruction in line 1 is equal to 7. Complexity is calculated based on Halstead complexity formula: (N2/n2) * (n1/2). In this formula, n1 is equal to number of distinct operators, n2 is equal number of distinct operands, N1 is equal to total number of occurrences of operators and N2 is equal to total number of occurrences of operands. In Factorial benchmark source code, complexity of line 6, The complexity is calculated as follows:

$$N2 = i, 1, i, num, i$$

$$n2 : i, 1, num$$

$$n1 : for(), =, ;, <=$$

The result of calculation is equal to 4. The Nesting level of an instruction shows the accessibility of the instruction. If the instruction is not in an *if* instruction, its nesting level is 0; if it is in an *if* instruction, then its nesting level is 1. For example, because of *else* and *for* instruction, the nesting level of 7 line of code is equal to 2.

4.2 Generating the Training Dataset

In this section of the paper, stage 3 of the proposed method (as shown in Fig. 5) is discussed. The required dataset to train the machine learning algorithm is prepared in stage 3. The identified features of the dataset are shown in Table 4 and then quantified using the generated CFG and real execution. There is a record (row) for each program instruction in the created dataset. As mentioned earlier, static features were quantified by means of source code and CFG. The average number of executions of each instruction was also measured as a dynamic feature by real executions. Table 8 (as a part of the generated dataset) shows the values of each feature for all the instructions of the factorial program. The columns of Table 8 indicate the following features, respectively: the number of variables defined in the instruction (static feature), the number of computational operators (static feature), the number of conditional operators (static feature), the number of used variables in the instruction (static feature), the number of numerical values used in the instruction (static feature), the number of data dependencies of the instruction (static feature), the number of conditional dependencies of the instruction (static feature), instruction length (static feature), instruction complexity (static feature), execution level of the instruction (dynamic feature), nesting level of each instruction (static feature), and the last feature error propagation rate of an instruction.

The last column (feature) is the dependent variable, and the other features are independent variables that are used in the training stage of the machine learning algorithm. An extensive series of mutation testing experiments has been performed in order to measure the error-propagation rate of each instruction. In these experiments, all possible bugs (mutants) have been injected into each instruction separately by the MuJava tool automatically [31]. After injecting a mutant (bug) into an instruction in a benchmark program, the program was executed 100 times by the selected coverage-based test data. Indeed, the error propagation rate of each instruction in a program has been measured by 100 executions in the presence of the injected mutant. Equation 1 is used to measure the error-propagation rate of an instruction. The number of times the program fails divided by 100 indicates the error propagation rate of an instruction. The MuJava is used to measure the error-propagation rate of each instruction and quantify the last feature. The features, listed in Table 8, are used in the form of a dataset for training the machine learning algorithms to create the instruction classifier.

Table 9 Created *Factorial* program dataset as the input of the ML algorithm

ANR	NVD	NOP	NCOP	NVU	NID	NDD1	NDD2	NCD	IL	IC	NL	RANK
1	1	0	0	0	0	0	0	0	8	2	0	C
1	1	0	0	1	0	0	0	0	4	1	0	C
1	0	0	1	1	1	0	0	0	5	1	0	C
0	0	0	0	1	0	0	0	1	5	1	1	D
1	0	0	1	0	0	1	1	1	2	1	0	B
5	2	1	1	2	1	1	1	0	13	4	1	A
5	1	1	0	2	0	2	0	1	6	2	2	B
1	0	0	0	2	0	1	0	0	9	1	1	D

$$\text{Error} - \text{Propagation Rate} = \frac{\text{Number of failures}}{\text{Total Number of Execution}} * 100\% \quad (2)$$

A larger data set has been used to check the performance of the data set for entering the machine learning algorithm Table 9. This data set is taken from *Factorial* program. Table 10 depicted dataset extracted from Maximum program as the input of ML algorithm. As shown in the Table 10, the RANK column indicates the program instruction type. In this example, the type D instructions are three instructions. The technique used in this paper studies this level of instructions. The source code of this program is shown in Fig. 8.

4.3 Creating the Instruction Classifier using Machine Learning Algorithm

The impetus behind this stage is to use machine learning (ML) to create a classifier for classifying program instructions according to the degree of error propagation. In this study, different machine learning (ML) algorithms (Gradient Boosted Trees, Decision Tree, Multi-Layer Perceptrons, and Random Forest) have been used to create an instruction classifier and the performance of the created classifier has been compared with each other. This section includes two stages: model training and model testing. In the training step of the proposed method, a program-instruction classifier is constructed by the ML algorithm as the base learning algorithm. In this stage,the authors used static analysis methods for converting a program into the identical executive version which is simpler and Lighter than the original program. It should be noted that determining the error-propagation rate of the data and instructions with helping the graphical representation is simpler than the main program. The features in the program instructions are considered as the input layer of ML algorithm. Create a dataset with 100 examples using the static and dynamic analysis of source-codes by setting a local random seed (default = 1992) to ensure repeatability. Converting the label attribute(rank) from polynomial to *RANK* using the appropriate operator enables us to select specific binominal classification performance measures. *Split data* into two partitions: an 80% partition (80 examples) for model building and validation and a 20% partition for testing. An important point to note is that data partitioning is not an exact science, and this ratio can change depending on the data. Connecting the 80% output (upper output port) from the *Split Data* operator to the *Split Validation* operator. Select a relative split with a ratio of 0.7 (70% for training) and shuffled sampling. Insert one of the *NN, DT, GBT and RF* operator in the Training panel of the *Split Validation* operator and the usual *Apply Model* operator in the Testing panel that embedded into *Optimize Parameters (Grid)*. Add a *Performance (Binomial Classification)* operator. Select the following options in the performance operator: accuracy, recall, precision, kappa and Root_Mean_Squared_Error. Figure 9 shows a view of the implementation of machine learning algorithms in rapidminer tool.

The structure of the required dataset for training the ML is explained in Table 8. In order to construct the classification model, each data set has been divided into K subsets and k-1 subsets are used as training data; to test the constructed model, the remaining subset is used. This process has been repeated k times. It should be noted that the training and test data should have the same distribution. To this end, the datasets are divided into training and test subsets in such a way that the training and test data include the same percentage of faulty and non-faulty instances. Finally, the constructed classifier takes the attributes of a program instruction and predicts its error-propagation rate category. The features of an instruction that are required to classify it are explained in Table 4. The assumed datasets were obtained for all the benchmark programs. The dataset is the input of the ML algorithm. In this study, the RapidMiner tool (https://rapidminer.com/) was used to implement the required ML algorithm. Table 11 depict the features of the input dataset. The created classifier classifies the instructions of the program based on their features. In this study, the program instructions are categorized into 4 categories by the created classifier; Table 10 illustrates the determined categories for the

Fig. 8 A source code of a Maximum program as case study 2

```
public static int Max(int[]x)
    {
1.      int n ;
2.      n=x.length;
3.      System.out.println(n+" is number of integers");
4.      int max=x [0];
5.      System.out.println(max+" is the first number  ");
6.      for(int i = 1;i<n;i+=1)  {
7.      if(max<x[i])
8.         max = x[i];
                }
9.      System.out.println("maximum of  " +n + "  numbers is  " +max);
10.     return max;
    }
```

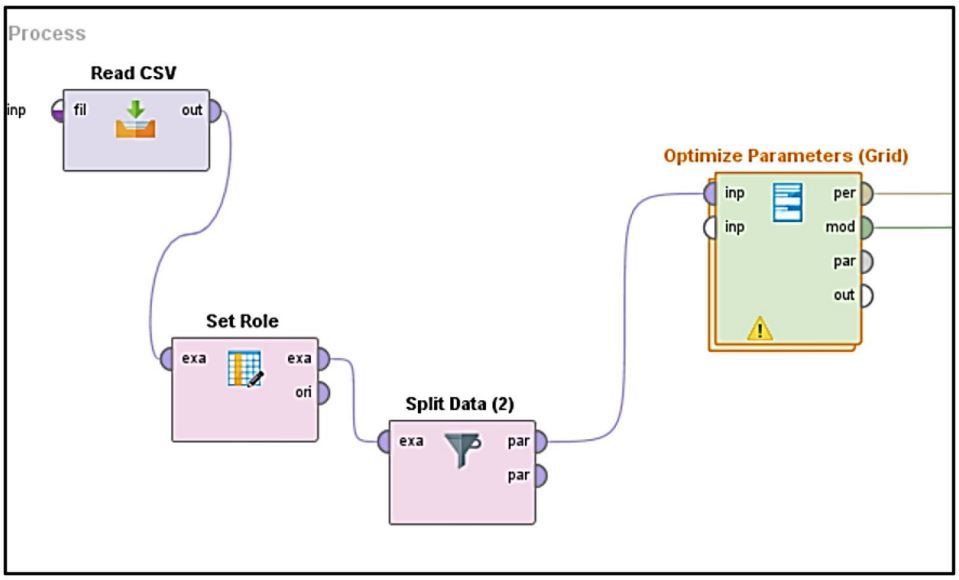

Fig. 9 A view of the implementation of machine learning algorithms in rapidminer

program instructions. After training the ML algorithm with the dataset, the created classifier should be tested. The training data was used to test in this study. The created instruction classifier was evaluated (tested) using the tenfold cross validation technique. As mentioned previously, the distribution of attributes in training and test data is 80% for training and 20% for testing.

4.4 Mutation Test

Stage 4 of the proposed method is the mutation test. Figure 10 shows the workflow of the fourth stage (mutation test) of the proposed method. In this stage, only the error-prone instructions (identified by the created classifier in the previous stage) are considered to be injected as mutants (bugs). This stage is implemented by the MuJava tool to automatically inject the bugs. The program instructions that are classified as the non-error-propagating instructions are avoided in the mutation test. Mutating the non-error-propagating instructions more likely generates equivalent mutants. In fact, the suggested method makes use of the developed classifier to identify the program's most error-propagating instructions. The overall number of mutants was decreased by avoiding the mutation of the non-error-propagating instructions. The mutation scores are also improved by lowering the number of equivalent mutants. The proposed method tries to reduce the number of generated affectless mutants (equivalent); to this end, after classifying the program instruction by the first step of the proposed method, the instructions in the class D are not considered for mutation in the second step. The mutation test was executed 5 times with 5 test suits for each benchmark program.

5 A Review of Mutation Test Environments

Many current applications provide high performance to process large volumes of data. These applications usually run in highly distributed environments Nevertheless, the large and complex architectures required for deploying

Table 10 *Maximum* program dataset as the input of the ML algorithm

LOC	ANR	NVD	NOP	NCOP	NVU	NID	NDD1	NDD2	NCD	IL	IC	NL	RANK
1	1	1	0	0	1	0	0	1	0	1	1	0	C
2	1	0	0	0	2	0	0	1	0	2	1	0	B
3	1	0	0	0	1	0	0	0	0	1	1	0	D
4	1	1	0	0	2	1	0	1	0	4	1	0	B
5	1	0	0	0	1	0	0	0	0	1	1	0	D
6	5	1	1	1	2	1	3	1	0	9	4	0	A
7	5	0	0	1	2	0	2	1	1	4	1	1	B
8	5	0	0	0	2	0	1	0	1	3	1	1	B
9	1	0	0	0	1	0	0	0	0	1	1	0	D
10	1	0	0	0	1	0	0	0	0	1	1	0	C

these applications may not be available during the development phase. Usually, these applications are tested against a small number of test cases that are manually designed by the testers. It is desirable to have effective test suites in order to detect failures in the application models. The basic characteristics of mutation testing tools include operators supported by the tool, mutation generation methods, and speed-up methods [31].

5.1 Mujava

MuJava [40] is a mutation testing system developed for the testing of Java programs. Its primary purpose has been to investigate mutation operators specific to object-oriented programming languages like Java. The prospect of using MuJava in a large-scale software development setting is an attractive one, as its mutation operators represent the state of the art in mutation testing research. However, as it is an experimental system, the extent to which it is scalable is unclear. The source code for MuJava is not publicly available, so its modification for improved scalability is difficult from a legal and practical point of view. Furthermore, its test format is not JUnit. It was infeasible for us to port our large legacy set of JUnit tests.

5.2 MuTomVo

MuTomVo was designed to be used in simulation tools based on OMNeT++ [50]. MuTomVo can be used with any other simulation tool based in OMNeT++. MuTomVo is a mutation testing framework which provides mechanisms to generate test suites and evaluate their effectiveness. MuTomVo allows to apply the mutation operators for reproducing the common mistakes made by competent programmers. MuTomVo is modular in the sense that its functionality is divided into independent modules. Four modules used to carry out mutation analysis: mutation engine, code analyzer, mutant builder and code generator. Consequently, different modifications can be applied to each module without interfering with the rest of the framework. On the other hand, this framework is flexible in the sense that different approaches can be integrated in each module. For instance, different code analyzers can be used to process the source code of different programming languages like, among others, C, C++ and Java.

5.3 Jester

Another tool that has been reviewed is Jester. Jester, in mutation testing, is considered an expensive tool for branch testing [34]. In fact, Jester provides a solution to develop a default set of jump operators, but the problems related to the efficiency and reliability of this tool remain unsolved. Additionally, the jump operators offered by Jester are not context-aware and often lead to broken code. The important point is that the Jester method is used to generate, compile and run unit tests against a mutation. This process is repeated for each mutation of the system under test and is therefore inefficient. Because of these disadvantages, Jester is not used in practical evaluations.

5.4 Jumble

Jumble has now been made available as an open source project on SourceForge at http://jumble.sourceforge.net/. The primary entry point to Jumble is a single Java class that takes as parameters a class to be mutation tested and one or more JUnit test classes. The output is a simple textual summary of running the tests. This is in the style of JUnit test output and includes an overall score of how many mutations were successfully caught as well as details about those mutations that are not. This includes the source line and the mutation performed. Variants of the output include a version compatible with Java development in emacs where it is possible to click on the line and go to the source line containing the mutation point [26].

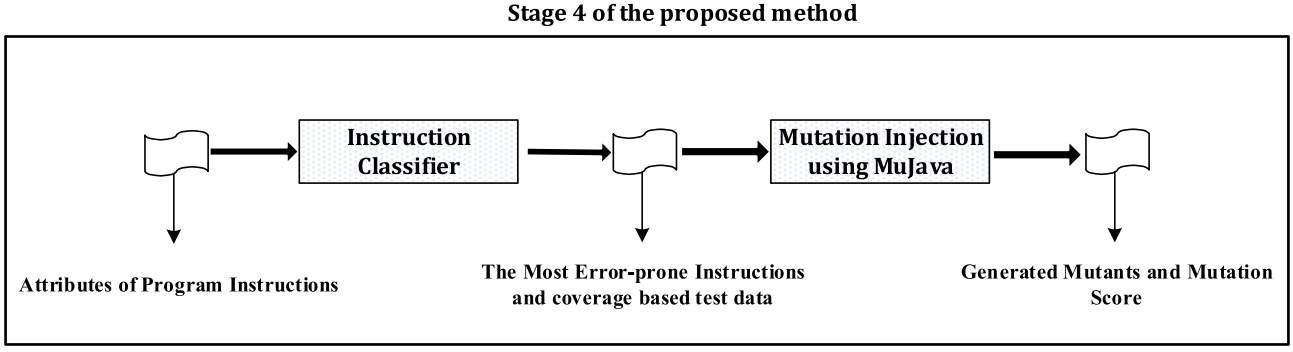

Fig. 10 The workflow of the fourth Stage of the Proposed Method

Table 11 The Features of the input dataset that was used to train the implemented ML algorithms in RapidMiner tool

ANR	NVD	NOP	NCOP	NVU	NID	NDD1	NDD2	NCD	IL	IC	NL	RANK
Average Num. of Runs	Num. of Variable Definition	Num. of Computational Operators	Num. of Conditional Operators	Num. of Variable usage	Num. of intermediate Data	Num. of Data Dependency type1	Num. of Data Dependency type2	Num. of Control Dependency	Instruction Length	Instruction Complexity	Nesting Level	RANK
					Independent Variables							Dependent Variable

5.5 Mothra

Mothra [17] is a mutation testing environment developed in the eighties for use with Fortran 77. While it cannot be used with Java software, it is the first fully featured mutation testing system and a large proportion of research on mutation testing described in the literature is based on it [37, 38, 40]. Mothra's mutation operators are the basis of what is done in Jumble.

6 Evaluation System

An extensive series of experiments have been performed to determine how well the recommended strategy reduces the number of mutants in the mutation test. A software-based experimental framework was developed to assess the effectiveness of the proposed approach. The steps of the experimental system are shown in Fig. 5. The software instructions were categorized using the newly developed classifier in the first stage. Predicting the error-propagation rate of each instruction in each benchmark program is the goal of this stage. The mutation test was run on program instructions with various error-propagation classes in the second stage of the research. A series of mutation test experiments were conducted to investigate the effectiveness of the proposed method as follows.

- In the first experiment, all of the benchmark programs' instructions get random mutation injections. In this experiment, the MuJava tool's default settings for injection site and timing are used.
- The instructions with the highest rate of error propagation are subjected to mutations in the second set of experiments. The instruction classifier produced by the ML algorithm recognized these instructions.
- Only the instructions with a reduced rate of error-propagation were subjected to mutation in the third series of experiments; the created classifier recognized these instructions.

After doing the above-mentioned experiments, the obtained results were compared with each other for the purpose of investigating and analyzing the effectiveness of the proposed method.

6.1 Benchmark Programs

To evaluate the effectiveness of the proposed method, six standard benchmark programs have been selected. These programs have been extensively used as standard

benchmarks in software testing research. Table 13 provides the brief of these programs. The many programming structures and operations used in real-world software are included in all of the benchmark programs, which were all created in the Java programming language. The chosen benchmarks are presented as functions (units). In fact, functions are present in the millions of lines of code that make up real-world applications. The standard size of a real-world function is between 5 and 50 lines of code. The benchmark programs are: *Triangle*, *Factorial*, *FindMax*, *Prime*, *Mid*, *Bubblesort* and *DOW*. The *Triangle* benchmark program was used to determine the type of triangle. The inputs of this program include 3 integers, which after checking, returns the output of the triangle type program, which includes equilateral, isosceles, and right-angled triangles. If the triangle formation condition is not met, the output will print an error message with the title of triangle formation failure. The *Factorial* program was used for computing the factorial of an integer. In the factorial program, the entered data value returns the factorial value of the entered number after checking the factorial condition in the output. The *Middle* program is used to compute and return the middle of three numbers. In this program, the input contains three numbers. In this program, the middle value of three entered numbers is calculated and returned as a result. The *FindMax* program specifies the maximum value of a list. *Bubblesort* takes a list of integer numbers and sorts them in ascending order. In the BubbleSort program, sorting is performed with variable number of inputs and the sorted string is printed in the output. The DOW program includes three categories of input named year, month and day according to the Gregorian date. After taking these three inputs, the program prints the equivalent day of the week of that date in the output. For example, the date 1986 month 4 day 18 is equivalent to Monday. Monday is printed as output.

The other required element in the conducted experiments is the test set. Random generation of test data does not guarantee the coverage of all the instructions of the program. Hence, in the conducted experiments, the coverage of all instructions should be taken into account. In simple terms, the extent to which the source code of a software program will get executed during testing is what is termed as *Code Coverage*. If the tests execute the entire piece of code including all branches, conditions, or loops, then we would say that there is complete coverage of all the possible scenarios and thus the Code Coverage is 100%. To understand this even better, let's take up an example. Figure 11 is a simple code that is used to add two numbers and display the result depending on the value of the result. This program takes in two inputs i.e. 'a' & 'b'. The sum of both is stored in variable c. If the value of c is less than 10, then the value of 'c' is printed else 'Sorry' is printed. Now, if we have some tests to validate the above program with the values of a & b such that the sum is always less than 10, then the else part of the code never gets executed. In such a scenario, we would say that the coverage is not complete.

Branch coverage aims at ensuring that every branch appearing in each conditional structure gets executed in source code. For instance, in the above code, all the 'If' statements and any accompanying 'Else' statement should all be covered by the test for a 100% Branch Coverage. For example, in the above code if value sets (2, 3), (4, 2), (1, 1) are used then Branch Coverage would be 100%. When data set (2, 3) is used then (b > a) and the first 'If' branch gets executed. Similarly, when data set (4, 2) is used then (a > b) evaluates to true and the second 'If' branch gets executed. Then with the data set (1, 1) the 'Else' branch evaluates to true and gets executed. Thereby, ensuring 100% branch coverage.

Statement Coverage is a measure that tells if all possible executable statements of code in source code have been executed at least once. It is a method to ensure that each line of the source code is covered at least once by the tests. This might sound simple but caution needs to be exercised while measuring the Statement Coverage. The reason being, in a source code there could be a certain condition that might not get executed depending on the input values. This would mean that all the lines of code would not be covered in testing. Thus, we may have to use different input value sets to cover all such conditions in the source code. For example, in the Fig. 12, if input values are taken as 2 & 3 then, the 'Else' part of the code would not get executed. However, if the input values are of type 3 & 2 then the 'If' part of the code would not get executed. This means that with either set of values of our Statement Coverage would not be 100%. In such a case, we may have to execute the tests with all three [(2, 3), (3, 2), (0, 0)] set of values to ensure 100% Statement Coverage.

For each benchmark program, different sets of test data have been prepared based on the coverage criteria. To this end, the CFG of each benchmark program is created. Edge coverage is used as a graph-based criterion to generate test data. Covering all edges in the CFG guarantees the coverage of the all-program source code. As shown in Fig. 7, covering all edges of the total CFG is considered as test criteria. In the second experiment, the CFG was created for each benchmark program after eliminating the non-error-propagating instructions of that program. In the final experiment, all instructions can be considered for coverage, but only the non-error-propagating instructions are important.

6.2 The Framework

In the first stage of the proposed method, visustin (http://www.aivosto.com/visustin.html) is used as a free software tool to generate the annotated CFG of the input source program. The generated CFG was used for static analysis of the program source code. The required dataset is generated using CFG and Tables 5 and 6. Tables 5 and 6 describe the DU-paths of the program variables. These tables are obtained from the source code and the related CFG. The created dataset in the form of a matrix (excel file) was used by the implemented ML algorithm in the RapidMiner tool set.

RapidMiner is a data mining tool with extensive data analysis libraries. The main element of the proposed method is an instruction classifier that was created by training the implemented ML algorithm. The classifier takes the features of an instruction (in the form of an array that is explained in Table 12) and determines its category in terms of error-propagation rate. The created classifier categorizes the instructions of a program into 4 categories (explained in Table 12). In the final stage, after classifying the program instructions, the mutation test is conducted only on the identified error-prone instructions. The mutation test is conducted using MuJava [31]. MuJava as a free mutation test tool is used in this study extensively. It automatically produces a set of mutants.

After generating mutant programs (buggy programs), Mujava uses the Junit tool for executing the test and evaluating the mutation score. As given in Table 14, mutation operators at the method level, supported by Mujava, are listed. After the CFG of each benchmark program is generated, the error propagation rate for each instruction is evaluated by the proposed method. Then, the instructions with a high error propagation rate are detected by the aid of the classifier which was obtained in the machine learning part. By changing every instruction in the benchmark programs, mutant programs (buggy programs) are created in the first evaluation stage. Only the instructions with a high error-propagation rate were subjected to mutation operators in the subsequent assessment stage. The instructions with a low rate of error-propagation are subjected to mutation operators in the third assessment stage. The status of the created mutants is then looked into in terms of being alive or killed. After that, it is calculated how many mutations were made on the instructions with the highest mistake propagation rate and the average mutation score. Finally, the obtained results are compared and contrasted with those of the previous related works.

Table 12 Categorizing the program instructions based on the error-propagation rate

propagation rate	Category
81%–100%	A
61%–80%	B
31%–60%	C
0%–30%	D

6.3 Results and Discussion

In this section, the statistical results obtained from the execution of tests and the generation of mutations and the results obtained for 6 benchmark programs are explained in detail.

6.3.1 Evaluating the Proposed Classifier

As explained in Section 4, the prepared dataset is used to train different ML algorithms. In this study, Gradient Boosted Trees (GBT), Decision Trees (DT), Deep Learning and Random Forest (RF) have been used as ML algorithms to create the desired instruction-classifier. The required dataset has been prepared in stage 2 of the proposed method. The same dataset was used to train the ML algorithm and the created classifier by the ML algorithms has been tested in the same way (k-fold). Table 15 shows the configuration operators for all 4 machine learning algorithms. Based on the value of each of these operators, the execution of the algorithm will be different. But by optimizing these operators, we have obtained the best implementation of the algorithms.

As explained in Section 4, the created classifier is a multi-class classifier; the outputs of the classifier are shown in Table 10. Every instruction in a 4-class classification must be sorted into one of four categories. Given a set of program instructions at the source code level, the generated classifier must determine which category (A, B, C, or D) each instruction belongs to. Indeed, the created classifier takes the features of an instruction and predicts its classes in terms of its error-propagation rate. The performance of machine learning classification is measured using the confusion matrix. Table 16 depicts the confusion matrix generated by the DT algorithm. The created classifiers by the ML algorithms were evaluated in terms of the following criteria.

- **Accuracy**: The first criterion for assessing classification models is their accuracy. Accuracy is the proportion of predictions made by our model that were accurate.
- **Precision**: The other performance criterion is precision. In multi-class classification, the precision is first calculated for each class, and then the average of the calculated values indicates the overall precision of the classifier. The precision for an instruction class is the number of correctly predicted instructions in the class out of all predicted instructions in the class.
- **Recall**: The recall as another criterion for an instruction class depicts the number of correctly predicted instructions in the class out of the total number of actual instructions into the class.

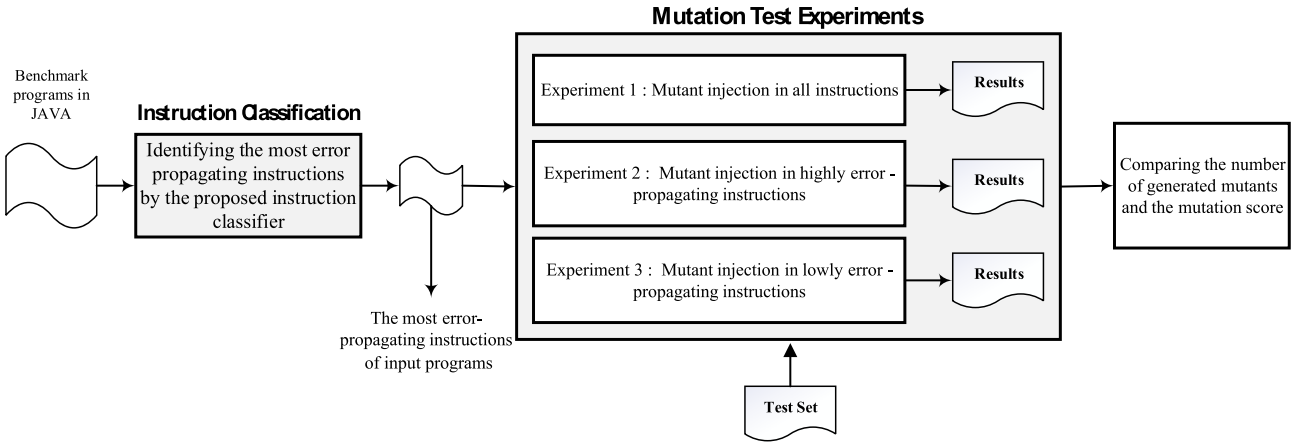

Fig. 11 Implemented experiments for evaluating the effectiveness of the proposed method in reducing the cost of mutation test

- **Kappa**: The kappa criterion is used to measure only those instructions that may have been correctly classified by chance. Kappa can be measured using both the total accuracy and the random accuracy.
- **RMSE**: RMSE is typically a performance metric for evaluation of a regression type machine learning model where a numerical label has been predicted. It can be found in the "Performance(Regression)" operator. If your dataset is a label and an associated prediction for each row, then it will be able to calculated using that operator.

This stage of the proposed method has been implemented in the RapidMiner tool set. RapidMiner includes an extensive data analysis library and it is one of the most frequently used tools for data analysis and data mining applications. Table 17 shows the performance of the created classifier by different ML algorithms in terms of accuracy, precision, recall, kappa and Root Mean Squared Error. ***Accuracy*** is the percentage of correctly classifies instances out of all instances. It is more useful on a binary classification than multi-class classification problems because it can be less clear exactly how the accuracy breaks down across those classes. ***Precision*** is the ratio of true positive samples to all samples classified as positive. It is also known as Positive Predictive Value (PPV). ***Recall*** is the ratio of true positive samples to all samples that are actually positive. It is also called True Positive Rate (TPR) or sensitivity. **Kappa** or Cohen's Kappa is like classification accuracy, except that it is normalized at the baseline of random chance on your dataset. It is a more useful measure to use on problems that have an imbalance in the classes (e.g. 70–30 split for classes 0 and 1 and you can achieve 70% accuracy by predicting all instances are for class 0). **RMSE** or Root Mean Squared Error is the average deviation of the predictions from the observations. It is useful to get a gross idea of how well

Table 13 Benchmarks programs characteristics

Programs	**Program size (code lines)**	**Program description**
Triangle	16	Determining triangle type
Factorial	8	Determining factorial number
Mid	10	Finding the middle of 3numbers
FindMax	15	Finding the greatest number
Prime	22	Determines if it is a prime number
Bubblesort	5	Sorting the list
DOW	56	Mapping day of week numeric to day of week

Table 14 The mutation operators that have been used by the MuJava tool in the proposed method

Operator	Description
AOR	Replacing the Arithmetic Operator
AOI	Inserting the Arithmetic Operator
AOD	Deleting the Arithmetic Operator
ROR	Replacing the Relational Operator
COR	Replacing the Conditional Operator
COI	Inserting the Conditional Operator
COD	Deleting the Conditional Operator
SOR	Replacing the Shift Operator
LOR	Replacement the Logical Operator
LOI	Inserting the Logical Operator
LOD	Deleting the Logical Operator
ASR	Replacing the Assignment Operato
Deletion operators added in 2013	
SDL	Deleting the Statement
VDL	Deleting the Variable
CDL	Deleting the Constant
ODL	Deleting the Operator

Table 15 Shows the configuration operators for all 4 machine learning algorithms

DT Algorithm	Decision Tree.criterion	Decision Tree.number_of_prepruning_alternatives	Decision Tree.apply_prepruning	Decision Tree.minimal_leaf_size	Decision Tree.maximal_depth
Value Range	*{ gain_ratio, information_gain, gini_index, accuracy}*	*[0,100]*	*{True,False}*	*[1, Infinity]*	*[-1,100]*
DL Algorithm	Deep Learning.adaptive_rate	Deep Learning.learning_rate_decay		Deep Learning.max_w2	
Value Range	*{True.False}*	*[0,1]*		*[0, 3.402823466385288 6E38]*	
GBT Algorithm				Gradient Boosted Trees.number_of_trees	Gradient Boosted Trees.min_split_improvement
Value Range				*[1,1000]*	*[0,1]*
RF Algorithm	Random Forest.confidence	Random Forest.criterion		Random Forest.apply_prepruning	
Value Range	*[1.0E-7,0.5]*	*{ gain_ratio, information_gain, gini_index, accuracy}*		*{True,False}*	

Table 16 Confusion matrix of the DT algorithm that is used to measure evaluation criteria

		Actual Classes			
		A	**B**	**C**	**D**
Predicted Classes	**A**	38	1	0	0
	B	0	14	0	0
	C	0	0	13	2
	D	0	0	0	12

(or not) an algorithm is doing, in the units of the output variable.

6.3.2 Evaluating the Mutation-Testing Method

The primary goal of this project is to lower the cost of software mutation testing by locating and removing effectless mutants (equivalent mutants). It is noteworthy that the static and dynamic analysis stages are aimed at accelerating and facilitating the next stages. A relationship was introduced for estimating the error-propagation rate of each instruction with respect to the CFG and source code. The ML algorithm classifies the highly error-propagating instructions of the program according to features. effectless mutants are determined according to the rate of propagation to the program output. Creating mutation in the instructions of source-code with a low propagation probability results in the ineffective mutation. Using ML classification, the proposed method intelligently finds a small subset of program code which includes maximum error-propagation rate. Then, rather than creating a large number of mutations in all source-code, only the instructions of error-propagating instructions are selected as the target instructions for creating mutations. Then the created mutations in the ineffective codes are ineffective and equivalent and removing these parts improves the efficiency of mutation test. The efficiency of the suggested strategy has been examined through a series of experiments. As shown in Table 13, a set of standard programs have been used as the benchmark programs. All the mutation test experiments have been performed on the MuJava and Junit platforms. For each benchmark program, five sets of test data were selected. Table 18 shows the generated coverage-based test data for the benchmark programs. These test sets were used during the mutation test. After generating the mutants of each benchmark program by MuJava, the mutant programs were executed by the selected test sets. The required metrics in this stage of the experiments are as follows.

- Number of generated mutants
- Number of killed mutants
- Number of live mutants
- The reduction rate of the live mutants
- The reduction rate of the total mutants

Table 17 The output of different ML in terms of accuracy, kappa, recall, precision and RMSE

Name of ML algorithm	Accuracy	Kappa	Recall	Precision	Root_Mean_Squared_Error
Random Forest (RF)	100%	1.000	100%	100%	0.280 ± 0.000
Decision Tree (DT)	96.25%	0.945	94.76%	95.91%	0.148 ± 0.000
Deep Learning	96.49%	0.948	95.45%	98.28%	0.237 ± 0.000
Gradient Boosted Trees (GBT)	96.25%	0.945	94.76%	96.03%	0.211 ± 0.000

At this stage of the proposed method, three sets of experiments have been performed separately on each benchmark program.

- All instructions of the benchmark program were mutated by the MuJava and then the generated buggy programs (mutants) were executed by the Junit using the prepared test data.
- Only the error-prone instruction identified by the proposed method were considered for the mutation in the MuJava. In this experiment, the generated mutants were executed by the Junit to evaluate the mutation score.
- Only the non-error-prorating instructions have been considered for the mutation test. Similar to the former experiments, the generated mutants were executed to calculate the mutation score.

Table 19 depicts the generated mutants for all benchmark programs. As seen in Table 19, all of the benchmark programs' instructions were subjected to MuJava operators (mutation operators). The discovered instructions by our developed classifier were regarded as mutations in the subsequent experiments. Class A instructions are the ones that propagate the errors to the program output; class D instructions, on the other hand, don't propagate the errors to the program output.

Figure 13 indicates the generated mutants for different instructions of each benchmark program by MuJava. On average, about 43.24% of the generated mutants are related to the most error-propagating instructions that were identified by the classifier. About 38.29% of the generated mutants are related to the instructions of classes B and C. About 18.45% of the generated mutants are related to the non-error-propagating instructions. The error-propagation rate of the instructions is not taken into account by the typical mutation testing methods, which operate in a brute-force way. In fact, all program instructions are tested in the mutation testing methods and tools. Consequently, a sizable portion of the created mutants are equivalent. Meanwhile, the proposed method performs the mutation operators only on the instructions that were classified in classes A, b, C; the instructions in class D (non-error propagating instructions) are avoided to mutate. The generated mutants on the non-error propagating instructions are more likely equivalent.

Figure 14 shows the total number generated mutants in each benchmark programs in two experiments. In the first experiment (brute force technique), all instructions of the programs were mutated by the mutation operators of the MuJava. On the second experiment, the non-error propagating instructions were eliminated in the mutation test and the instructions in classes A, B, and C (error-propagating instructions) were mutated. In every experiment, the suggested method (error-propagation aware method) produces fewer mutants than brute force mutation testing. The average number of mutants produced by the brute force mutation in MuJava is around 136.28, but the average number of mutants produced by the suggested approach is approximately 114.42. In fact, the number of mutants for a program function (unit) produced by the suggested procedure is far smaller than the number of mutants produced by the brute

Table 18 Five different test set for each benchmark that were selected based on the code coverage criteria

Programs	Num. of inputs	Test Suit
Triangle	3	{(12, 12, 18), (14, 14, 14), (5, 4, 3), (13, 14, 13), (8, 6, 10)}
Factorial	1	{20, 15, 17, 3, 2}
Mid	3	{(231, 231, 231), (14, 12, 19), (180, 643, 522), (200, 200, 120), (213, 421, 213)}
FindMax	5	{(453, 564, 87, 235, 78), (45, 64, 193, 77, 78), (128, 93,14, 31, 0), (93, 432, 37,11, 42), (89, 75, 63, 142, 29)}
Prime	1	{17, 21, 23, 27, 19}
Bubblesort	List of *n* integer values	{(25, 26, 87, 230, 298, 1000, 1020), (1,-3,19,77,0, 658, 9), (35, 30, 29, 21, 15, 8, 7), (93, 432, 108, 370, 42, 93), (9, 75, 603, 12, 9, 2)}
DOW	3	{(14,1,1994),(22,1,1962),(9,11,1904),(1,2,1940),(15,5,1955)}

Table 19 Generated mutants for all benchmarks by the mujava tool

program name	Total mutants	Number of mutants for a instructions	Number of mutants for b instructions	Number of mutants for c instructions	Number of mutants for d instructions
Triangle	252	96	79	33	44
Bubble sort	97	52	23	12	10
Factorial	68	27	15	8	18
Prime	66	29	12	10	15
Middle	121	58	28	20	15
FindMax	56	19	10	9	18
DOW	294	107	101	52	34

force mutation test. The results show that the suggested strategy significantly reduced the number of mutants, which in turn reduced the cost of the mutation test. The selected test suit based on code coverage for each benchmark program has been explained in Table 18.

Figure 15 illustrates the number of live mutants of seven programs in the five executions by the Junit. The live mutants are the mutants that were not killed (detected) by the test sets. On average, the number of live mutants in the brute force methods (traditional methods) is shown with green color for all programs; for Triangle program, the number of live mutants in the proposed method is about 107. Indeed, 29.13% of the live mutants have been reduced by the proposed method in the test of the *Triangle* program. The proposed method tries to reduce the equivalent mutants by avoiding the mutation of the non-error-propagating instructions. The proposed method has similar effectiveness to the *bubblesort* benchmark program. The number of live mutants is considerably reduced by the proposed method. On average, the number of live mutants in the *bubblesort* program is about 3.4, whereas in brute force mutation testing this figure is about 9.4.

The results of the mutation testing on the *factorial* program have been shown in Fig. 15 using the five different test sets. The results demonstrate that the number of live mutants may be significantly decreased using the suggested strategy.

Input a, b

Let c = a + b

If c < 10, print c

Else, print 'Sorry'

Fig. 12 A simple program used to illustrate code coverage

The brute force mutation test produces an average of around 23.2 live mutants; the suggested approach produces an average of about 9.4 live mutants.

Similar mutation experiments have been performed on the *Prime* benchmark program. Firstly, all instructions in the program were mutated by MuJava. Indeed, all mutation operators of the MuJava performed in all instructions of the program in an automatic platform. Secondly, the mutation operators were performed only on the error-propagation instructions that had been identified by the first step of the proposed method. In the Prime, the proposed method provides similar results. Figure 14 illustrates how fewer affectless mutants were produced using the proposed method. The *Middle* program is another benchmark that is applied in the tests. This program offers several program structures that may be utilized in real-world projects. In fact, the chosen benchmark programs perform features that are common to real-world systems. The number of live mutants created using the suggested method is less than those generated using the brute force method, as illustrated in Fig. 15. The similar experiment was performed on the *Max* program. Similarly, to the previous experiments, the affectless mutants were avoided, and consequently the mutation score of the selected test sets has been increased. All in all, the proposed method is successful in reducing affectless mutants.

The final experiment in Fig. 15 was performed on the *DOW* program. This program is bigger in number of lines(56 LOC) compared to the previous programs. With similar results observed, comparing the number of live mutations compared to the original program shows success in reducing live mutations. Figure 16 shows mutation score in the mutation test performed on the all programs in the brute force and error-propagation aware (proposed) methods with 5 test sets. Eliminating the ineffective mutants (equivalent mutants) causes an increase in the mutation score of the test sets. The results show the effectiveness of the proposed method in terms of the ineffective mutants' reduction. As shown in Fig. 16, the mutation score of all test sets has been increased in the error-propagation aware mutation test. In the

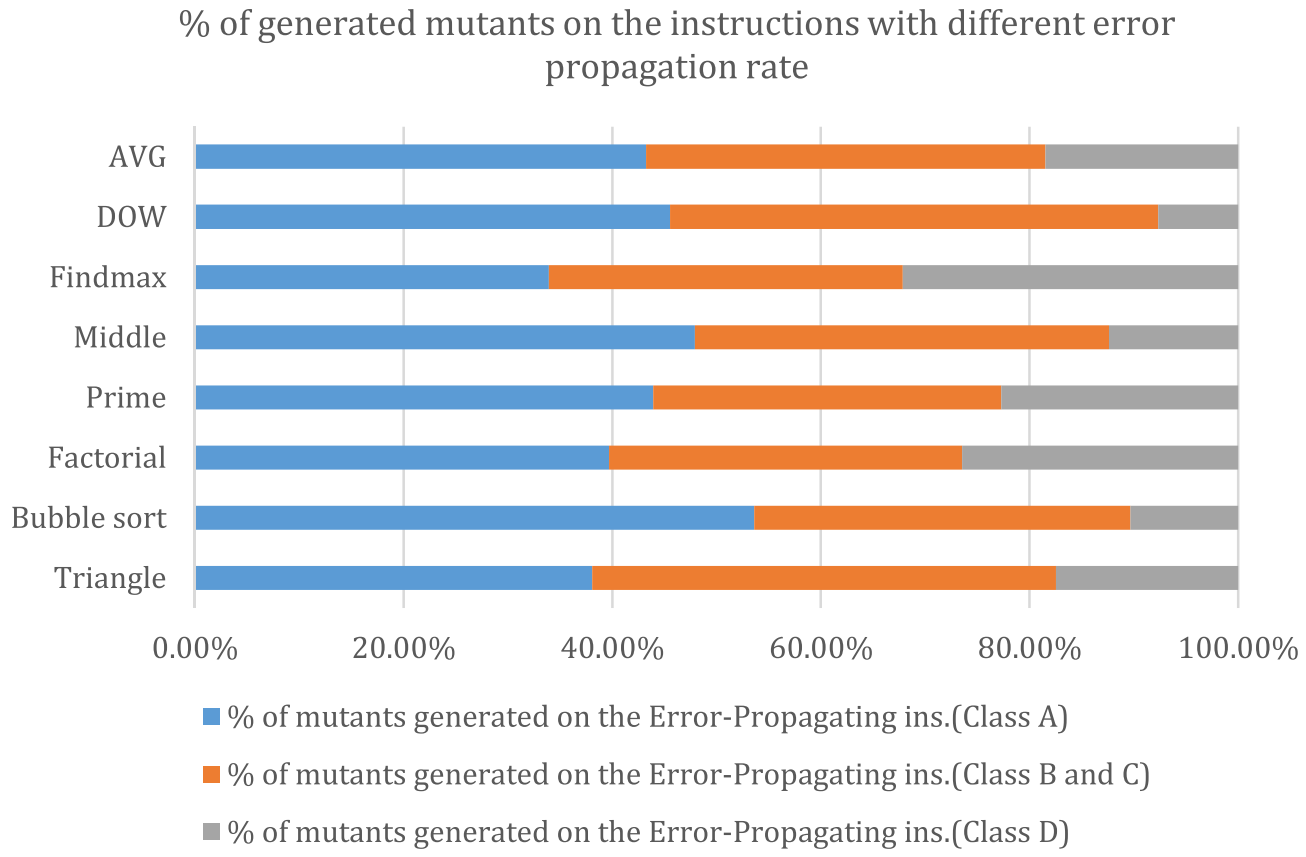

Fig. 13 Percentage of the generated mutants on the different instructions with different error propagation rate by MuJava

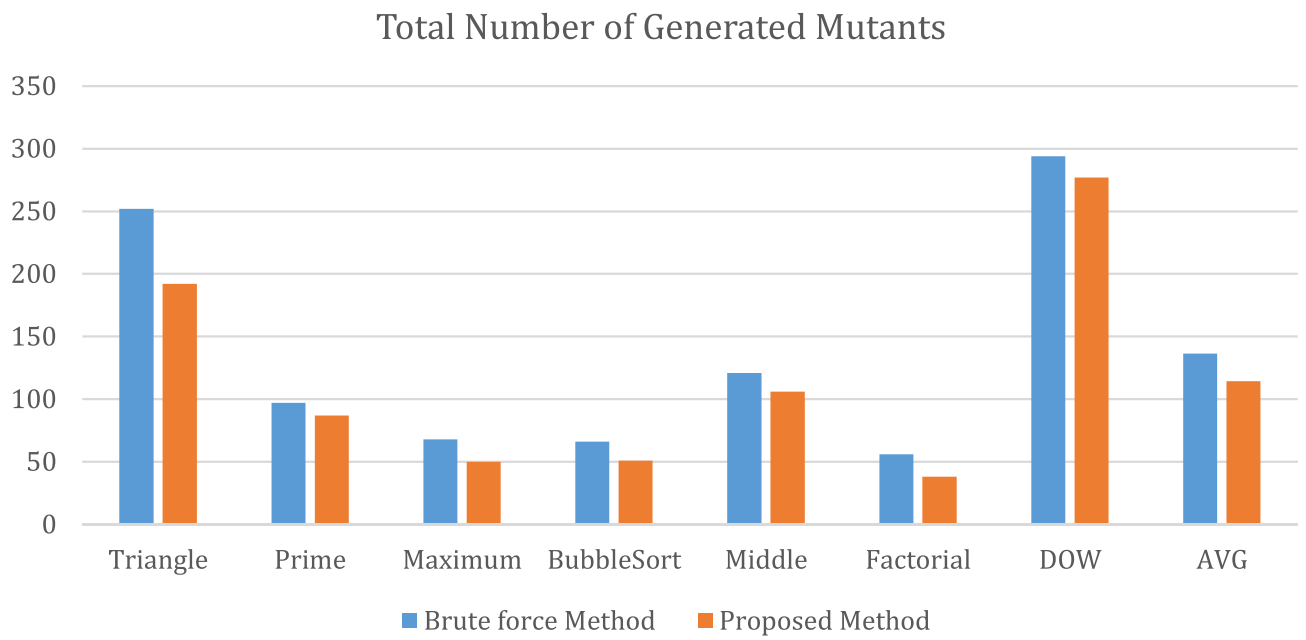

Fig. 14 Number of generated mutants in two brute force and error-propagation aware (proposed) methods

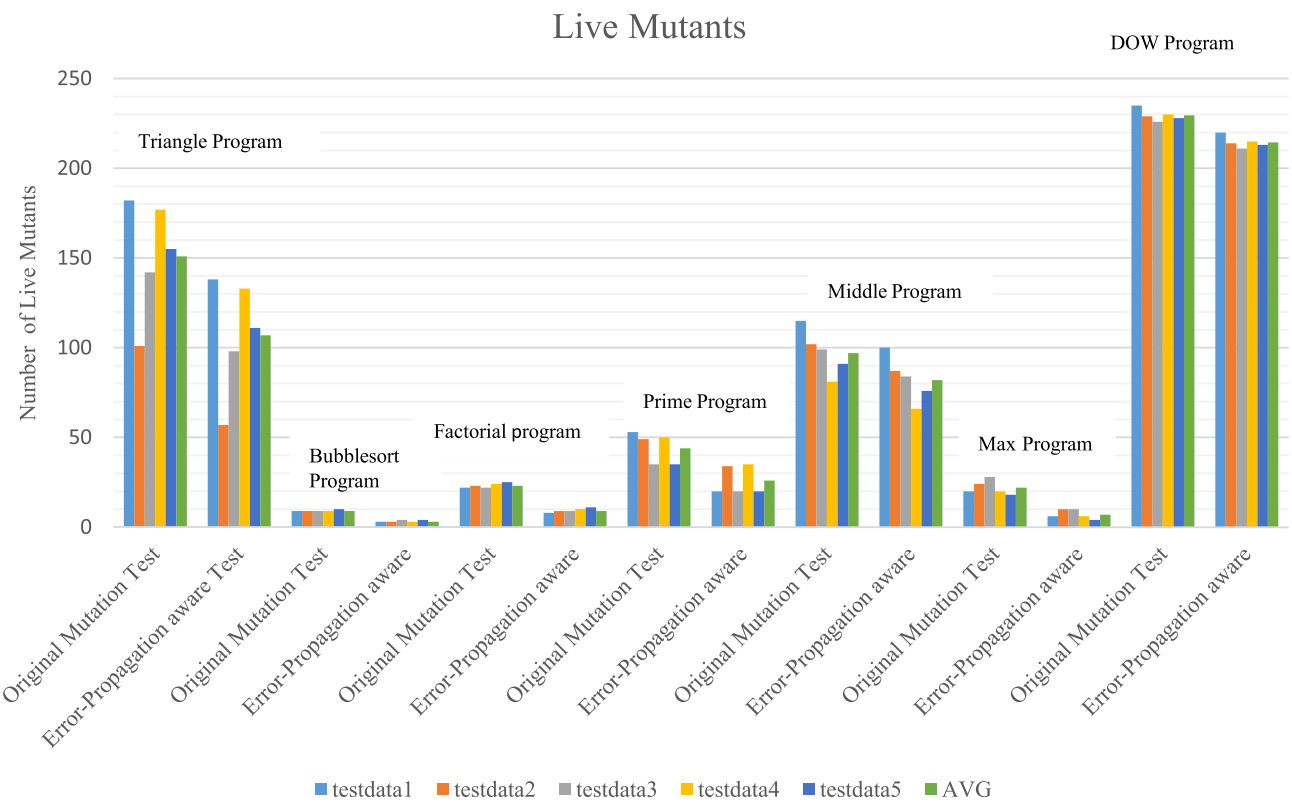

Fig. 15 Number of Live mutants in the mutation test performed on the all programs in the brute force and error-propagation aware (proposed) methods with 5 test sets

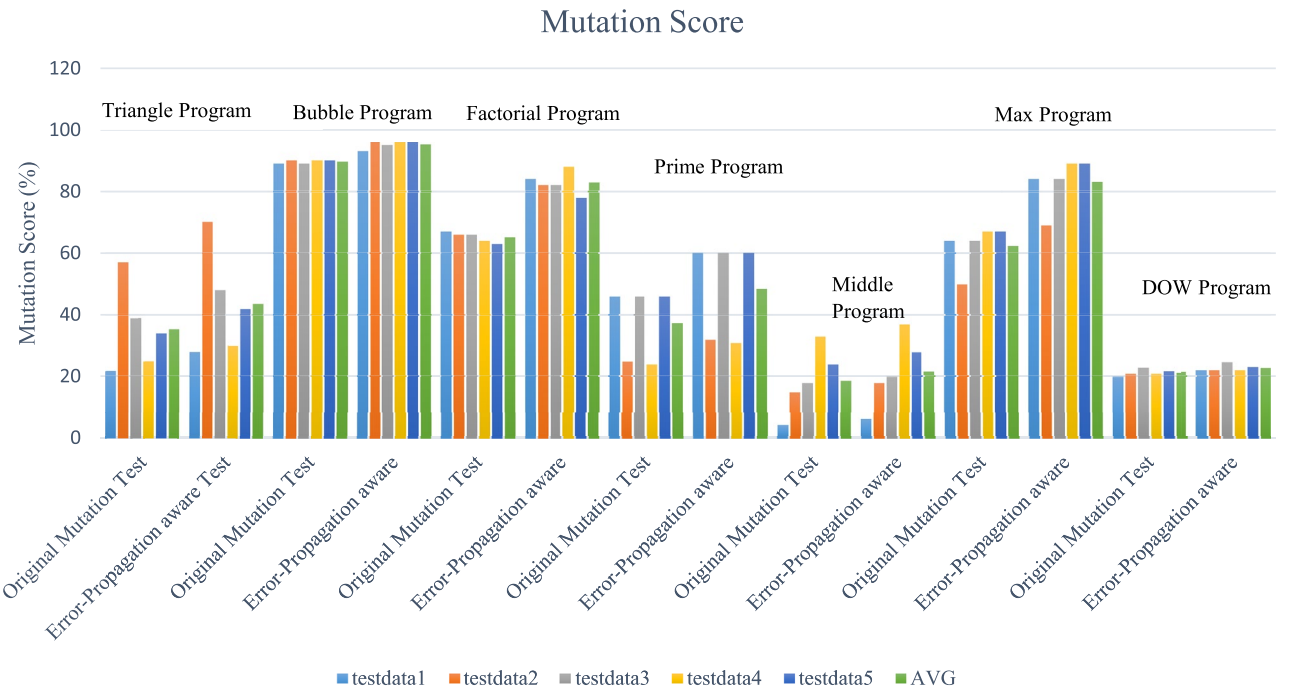

Fig. 16 Obtained mutation score in the mutation test performed on the all programs in the brute force and error-propagation aware (proposed) methods with 5 test sets

Table 20 The number of generated mutants for all benchmarks in brute-force and proposed method by the MuJava tool

Benchmark	Brute force Method	Proposed Method	Percent of the reduced total mutants
Triangle	252	192	13.12%
Prime	97	87	10.30%
Maximum	68	50	26.47%
BubbleSort	66	51	22.72%
Middle	121	106	12.39%
Factorial	56	38	32.14%
DOW	294	277	5.78%

error-propagation aware mutation test, the obtained increment in the mutation score of the same test sets is about 25.70%. Additionally, the suggested method outperforms the brute force method in terms of mutation score for the identical test sets. Additionally, the suggested method outperforms the conventional method in terms of mutation score across all test sets (five test sets). The mutation score of a test set can be evaluated accurately with a limited number of mutants.

As shown in Fig. 16, the mutation score of the test sets for *bubblesort* program reached 95.20% in the proposed method. The mutation score of the identical test sets therefore increased to 82.8 percent. Therefore, the *factorial* experiments' findings show that the suggested method is effective at identifying and removing affect-less mutants. For *Middle* program, the suggested method outperforms the conventional method in terms of mutation score across all test sets (five test sets). Similarly, to the previous experiments, for *Max* program, the mutation score of a test set can be evaluated accurately with a limited number of mutants. The final experiment was performed on the *DOW* program. The average value of the test score in the *Dow* program for the original program is about 21.4 and for error-propagation aware method is 22.7. This increase is a very significant number for a program with 56 lines of code.

Table 21 The average number of live mutants in five test executions in brute-force and proposed method by the MuJava tool

Benchmark	Brute force Method	Proposed Method	Percent of the reduced live mutants
Triangle	151.4	107.4	29.06%
Prime	44.4	25.8	41.89%
Maximum	22	7.2	67.27%
BubbleSort	9.4	3.4	63.82%
Middle	97.6	82.6	15.36%
Factorial	23.2	9.4	59.48%
DOW	228	213	6.57%

Tables 20 and 21 indicate the percent of the reduced total and live mutants in all benchmark programs in the proposed method. The number of produced mutants is typically reduced by roughly 19% using the suggested method. Additionally, the live mutants are reduced by 32.24% as a result of the suggested technique. It should be noted that the suggested method tries to eliminate only the effect-less mutant (equivalent). The key technical benefit of the suggested method is that the mutation of the instructions that don't propagate errors is avoided. These findings could help the current mutation-test tools and procedures perform better. The method described in this study may be used with MuJava, Muclipse, and other related mutation tools. Finally, researchers and software developers can benefit from the results of this study.

7 Conclusion

To sum it up, the proposed method was divided into two significant phases. In the first phase, the source code of the input-program source code was statically and dynamically investigated to identify the error-propagating features of the program instructions. With regard to the identified features, a dataset was created using a set of standard benchmark programs. In the next phase of the method, the dataset was used to train the ML algorithms and create an instruction classifier. The constructed classifier classes the program instructions based on its error-propagating rate. The instructions that are classified as error-propagating, are considered in the mutation testing by the MuJava tool. The non-error-propagating instructions are avoided due to mutation in the proposed method. This technique aims to reduce the number of mutants generated, specifically targeting non-error propagating instructions, in order to improve the efficiency of mutation testing. By selectively excluding non-error propagating instructions from the mutation process, the proposed technique effectively reduces the number of mutants generated during mutation testing. The proposed method identifies the strategic locations of the program source code in terms of error-propagation rate and performs the standard mutation operators on them. Avoiding the non-error propagating instructions caused a considerable reduction in the number of generated mutants in the mutation test. Consequently, a sizable portion of the created mutants are equivalent. Meanwhile, the proposed method performs the mutation operators only on the instructions that were classified in classes A, b, C; the instructions in class D (non-error propagating instructions) are avoided to mutate. The generated mutants on the non-error propagating instructions are more likely equivalent. each benchmark

programs in two experiments. In the first experiment (brute force technique), all instructions of the programs were mutated by the mutation operators of the MuJava. On the second experiment, the non-error propagating instructions were eliminated in the mutation test and the instructions in classes A, B, and C (error-propagating instructions) were mutated. In every experiment, the suggested method (error-propagation aware method) produces fewer mutants than brute force mutation testing. Therefore, the proposed method of reducing mutation testing focuses on selectively mutating error-propagating instructions while avoiding non-error propagating instructions. The results show that the suggested strategy significantly reduced the number of mutants, which in turn reduced the cost of the mutation test.

The proposed approach for reducing mutation testing involves selectively applying mutation operators on error-propagating instructions, while avoiding non-error propagating instructions. One of the main problems associated with mutation testing is the large number of equivalent mutants that are generated. The strategy of selectively applying mutation operators on error-propagating instructions and avoiding non-error propagating instructions in the proposed approach helps to reduce the number of equivalent mutants. In the one of the previous methods in term of mutation reduction, using the *genetic algorithm*, paths with a high impact on the error propagation rate have been identified, and mutants were only applied to the paths with a high error propagation rate [24]. But the problem of this method is the misclassification of killable mutants instead of equivalent mutants and the misclassification of equivalent mutants instead of killable mutants. although in this method, instead of examining the program at the level of instructions, the program was evaluated at the level of the path, and this caused low efficiency in examining the mutants. Because the granularity of program evaluation in the genetic algorithm method is larger than the proposed method. Finding the error-propagating instructions of a program source code can be mapped in an optimization problem and then it can be solved by different metaheuristic algorithm suggested in [4, 7, 55]. Error propagation rate is an indirect metric of an instruction in a program. some of them have been identified in this study, but identifying the other effective features is suggested as one of the future studies. The other extension of this study is using deep learning to create more accurate and more precise classifiers.

Author Contribution All authors contributed to this study.

Funding The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Data Availability The data related to the current study is available in the google. Drive and can be freely accessed by the following link: https://drive.google.com/drive/folders/1QqFFXYpgintZeSgQi7uzytBDPdkNpuBD?usp=sharing

Declarations

Conflict of Interest The authors have no relevant financial or non-financial conflict interests.

References

1. Acree A, Budd T, DeMillo R, Lipton R, Sayward F (1980) Mutation Analysis, School of Information and Computer Science. Georgia Inst Technol
2. Arasteh B (2018) Software Fault-Prediction using Combination of Neural Network and Naive Bayes Algorithm. J Netw Technol 9(3):94–101. https://doi.org/10.6025/jnt/2018/9/3/94-101
3. Arasteh B (2019) ReDup: A software-based method for detecting soft-error using data analysis. In Comput Electr Eng 78(9):89–107
4. Arasteh B, Fatolahzadeh A, Kiani F (2022) Savalan: Multi objective and homogeneous method for software modules clustering. J Softw Evol 34(1):2022. https://doi.org/10.1002/smr.2408
5. Arasteh B, Miremadi SG, Rahmani AM (2014) Developing Inherently Resilient Software Against Soft-Errors Based on Algorithm Level Inherent Features. J Electron Test 30(9):193–212. https://doi.org/10.1007/s10836-014-5438-8
6. Arasteh B, Pirahesh S, Zakeri A, Arasteh B (2014) Highly Available and Dependable E-learning Services Using Grid System. Procedia Soc Behav Sci 143(2014):471–476. https://doi.org/10.1016/j.sbspro.2014.07.519
7. Arasteh B, Razieh S, Keyvan A (2020) ARAZ: A software modules clustering method using the combination of particle swarm optimization and genetic algorithms. Intell Decis Technol 14(4):449–462. https://doi.org/10.3233/idt-200070
8. Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for C. Softw Test Verif Reliab 11(2):113–136
9. Binu Rajan MR, Vinod Chandra SS (2017) ABC Metaheuristic Based Optimized Adaptation Planning Logic for Decision Making Intelligent Agents in Self Adaptive Software System. Lect Notes Comput Sci 10387:496–504
10. Bishop CM (1995) Neural networks for pattern recognition. Clarendon Press, Oxford, England. Oxford University Press, Inc. New York, NY, USA ©1995 ISBN: 0198538642. Available at: http://cs.du.edu/~mitchell/mario_books/Neural_Networks_for_Pattern_Recognition_-_Christopher_Bishop.pdf
11. Breiman L (2001) Random forests. Mach Learn 45(1):5–32
12. Budd TA (1980) Yale University, Mutation Analysis of Program Test Data
13. Bouyer A, Arasteh B, Movaghar A (2007) A new hybrid model using case-based reasoning and decision tree methods for improving speedup and accuracy. IADIS International conference of applied computing
14. Chandra SV, Sankar SS, Anand HS (2022) Smell detection agent optimization approach to path generation in automated software testing. J Electron Test 38(6):623–636. https://doi.org/10.1007/s10836-022-06033-8
15. Dang X, Gong D, Yao X, Tian T, Liu H (2022) Enhancement of Mutation Testing via Fuzzy Clustering and Multi-Population Genetic Algorithm. IEEE Trans Softw Eng 48(6):2141–2156

16. Delgado-Pérez P, Medina-Bulo I (2018) Search-based mutant selection for efficient test suite improvement: Evaluation and results. Inf Softw Technol 104(2018):130–143
17. DeMillo RA, Spafford EH (1986) The Mothra software testing environment, presented at The 11th NASA Softw Eng Lab Workshop Goddard Space Center
18. Deng L, Offutt J, Ammann P, Mirzaei N (2017) Mutation operators for testing Android apps. Inf Softw Technol 81:154–168
19. Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29(5):1189–1232
20. Friedman JH (2002) Stochastic gradient boosting. Comput Stat Data Anal 38(4):367–378
21. Ghaemi A, Arasteh B (2020) SFLA-based heuristic method to generate software structural test data. J Softw Evol Proc 32:e2228. https://doi.org/10.1002/smr.2228
22. Gheyi R, Ribeiro M, Souza B, Guimarães M, Fernandes L, d'Amorim M, Alves V, Teixeira L, Fonseca B (2021) Identifying method-level mutation subsumption relations using Z3. Inf Softw Technol 132:106496
23. Good IJ (1951) Probability and the Weighing of Evidence, Philosophy Volume 26, Issue 97, 1951. Published by Charles Griffin and Company, London 1950. Copyright © The Royal Institute of Philosophy 1951, pp. 163–164. https://doi.org/10.1017/S0031819100026863. Available at Royal Institute of Philosophy website: https://www.cambridge.org/core/journals/philosophy/article/probability-and-the-weighing-of-evidence-by-goodi-j-london-charles-griffin-and-company-1950-pp-viii-119-price-16s/7D911224F3713FDCFD1451BBB2982442
24. Hosseini S, Arasteh B, Isazadeh A, Mohsenzadeh M, Mirzarezaee M (2021) An error-propagation aware method to reduce the software mutation cost using genetic algorithm. Data Technologies and Applications 55(1):118–148. https://doi.org/10.1108/DTA-03-2020-0073
25. Howden WE (1982) "Weak mutation testing and completeness of test sets." IEEE Trans Softw Eng 8(4):371–379
26. Irvine SA, Pavlinic T, Trigg L, Cleary JG, Inglis S, Utting M (2007) Jumble Java byte code to measure the effectiveness of unit tests. Proceedings of the Test: Acad Ind Proc Pract Res Tech - MUTAT (TAICPART-MUTATION '07). IEEE Computer Society, USA, pp 169–175. https://doi.org/10.1109/taic.part.2007.38
27. King KN, Offutt AJ (1991) A Fortran language system for mutation-based software testing. Softw: Pract Exper 21(7):685–718
28. Kintis M, Papadakis M, Malevris N (2010) Evaluating mutation testing alternatives: a collateral experiment. Proc 17th Asia-Pacific Softw Eng Proc (APSEC)
29. Kurtz B, Ammann P, Delamaro M, Offutt J, Deng L (2014) Mutant subsumption graphs. 2014 IEEE Seventh Int Proc Softw Test Verif Valid Workshops (ICSTW)
30. Kurtz B, Ammann P, Offutt J (2015) Static analysis of mutant subsumption. IEEE Eighth Int Proc Softw Test Verif Valid Workshops (ICSTW)
31. Ma YS, Offutt J, Kwon YR (2006) MuJava: A Mutation System for Java. In 28th Int Proc Softw Eng (ICSE '06)
32. Malevris N, Yates D (2006) The collateral coverage of data flow criteria when branch testing. Inf Softw Technol 48(8):676–686
33. McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. Bull Math Biophys 5(4):115–133
34. Moore I (2001) Jester - a JUnit test tester
35. Mresa ES, Bottaci L (1999) Efficiency of mutation operators and selective mutation strategies: an empirical study. Softw Test Verif Reliabil 9(4):205–232
36. Nilsson NJ (1965) Learning machines. New York: McGraw-Hill. Published in: J IEEE Trans Inf Theory 12(3):407, 1966. Inf Theory 12(3), 1966. https://doi.org/10.1109/TIT.1966.1053912. Available at ACM digital library website: http://dl.acm.org/citation.cfm?id=2267404
37. Offutt AJ, Lee SD (1994) An empirical evaluation of weak mutation. IEEE Trans Softw Eng 20(5):337–344
38. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. Proceedings of the 15th Int Proc Softw Eng, ICSE '93, IEEE Computer Society Press, Los Alamitos, CA
39. Offutt J, Lee A, Rothermel G, Untch RH, Zapf C (1996) An Experimental Determination of Sufficient Mutant Operators. ACM Trans Softw Eng Methodol 5:99–118
40. Offutt J, Ma Y-S, Kwon YR (2006) MuJava: an automated class mutation system. Softw Test Verif Reliab 15:97–133
41. Osisanwo FY, Akinsola JET, Awodele O, Hinmikaiye JO, Olakanmi O, Akinjobi J (2017) Supervised machine learning algorithms: classification and comparison. Int J Comput Trends Technol (IJCTT) 48(3):128–138
42. Papadakis M, Malevris N (2010) An empirical evaluation of the first and second order mutation testing strategies. Third Int Proc Softw Test Verif Valid Workshops (ICSTW)
43. Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106
44. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back propagating errors. Nature 323(6088):533
45. Sharma B, Girdhar I, Taneja M, Basia P, Vadla S, Srivastava PR (2011) Software coverage: A testing approach through ant colony optimization. Lecture notes in computer science, vol 7076. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27172-4_73
46. Sommerville I (2018) Software engineering, 10th edn. Pearson India (ISBN: 9332582696)
47. Sridharan M, Siami-Namin A (2010) Prioritizing mutation operators based on importance sampling. In: Proceedings of the IEEE 21th Int Sympo- sium Softw Reliab Eng (ISSRE). IEEE, San Jose, CA, USA, pp. 378–387
48. Taiwo OA (2010) Types of Machine Learning Algorithms, New Advances in Machine Learning, Yagang Zhang (Ed.), ISBN: 978-953-307-034-6, InTech, University of Portsmouth United Kingdom. Pp 3 – 31. Available at InTech open website: http://www.intechopen.com/books/new-advances-inmachine-learning/types-of-machine-learning-algorithms
49. Uddin S, Khan A, Hossain ME, Moni MA (2019) Comparing different supervised machine learning algorithms for disease prediction. BMC Med Inform Decis Mak 19(1):1–16. https://doi.org/10.1186/s12911-019-1004-8
50. Varga A (2001) Discrete event simulation system. In Proc Eur Simul Multiconference (ESM'2001) (pp. 1–7)
51. Wei C, Yao X, Gong D, Liu H (2021) Spectral clustering based mutant reduction for mutation testing. Inf Softw Technol 132:106502
52. Wong WE (1993) On mutation and data flow. Ph.D. dissertation, Purdue University
53. Woodward M, Halewood K (1998) From weak to strong, dead or alive? An analysis of some mutation testing issues. Proc Second Workshop Softw Test Verif Anal
54. Yao X, Zhang G, Pan F, Gong D, Wei C (2022) Orderly Generation of Test Data via Sorting Mutant Branches Based on Their Dominance Degrees for Weak Mutation Testing. In IEEE Trans Softw Eng 48(4):1169–1184
55. Zadahmad M, Arasteh B, YousefzadehFard P (2011) A pattern-oriented and web-based architecture to support mobile learning software development. Procedia Soc Behav Sci 28(2011):194–199. https://doi.org/10.1016/j.sbspro.2011.11.037

56. Zhang L, Gligoric M, Marinov D, Khurshid S (2013) Operator-based and random mutant selection: better together. Autom Softw Eng (ASE),. IEEE/ACM 28th International Proc
57. Zhang L, Hou S-S, Hu J-J, Xie T, Mei H (2010) Is operator-based mutant selection superior to random mutant selection? Proceedings of the 32nd ACM/IEEE Int Proc Softw Eng, 2010

Zeinab Asghari is a researcher in in the science and research branch of islamic azad university. Her research interest includes software testing, evolutionary algorithms and their function in software dependability engineering.

Bahman Arasteh was born in Tabriz. He received the master degree in software engineering from Azad University of Arak, and the PhD degree in software engineering from Islamic Azad University, Tehran Science and Research Branch, respectively. Currently, he is an associate professor at Istinye University, Istanbul, Turkiye. He has published more than 50 papers in refereed international journals and conferences. He is the coordinating editor in the springer journal of electronic test and springer journal of system assurance engineering and management. He is the reviewer of different international journals in Elsevier, Springer, Wiley and Hindawi. His research interests include search-based software engineering, Software testing, optimization algorithms, software fault tolerance, and software security.

Abbas Koochari is an associate professor in the science and research branch of islamic azad university. His research interest includes search-based computer engineering, complex networks, optimization problems and meta heuristic algorithms.