IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SCALABLE DEEP LEARNING FOR BIG DATA

Received October 5, 2019, accepted October 21, 2019, date of publication October 29, 2019, date of current version November 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2950171

Scalable Mutation Testing Using Predictive Analysis of Deep Learning Model

MUHAMMAD RASHID NAEEM[1], TAO LIN[1], HAMAD NAEEM[1], FARHAN ULLAH[1], AND SAQIB SAEED[2]

[1]College of Computer Science, Sichuan University, Chengdu 610065, China
[2]Department of Computer Information Systems, College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Dammam 34212, Saudi Arabia

Corresponding author: Tao Lin (lintao@scu.edu.cn)

This work was supported in part by the Science and Technology Planning Program of Sichuan University and Luzhou under Grant 2017CDLZG30, in part by the Postdoctoral Science Fund of Sichuan University under Grant 2019SCU12058, and in part by the 2018–2020 Higher Education Talent Training Quality and Teaching Reform Project of Sichuan Province under Grant JG2018-46.

ABSTRACT Software testing plays a crucial role in ensuring the quality of software systems. Mutation testing is designed to measure the adequacy of test suites by detecting artificially induced software faults. Despite their potential, the expensive cost and the scalability of mutation testing with large programs is a big obstacle in its practical use. The selective mutation has been widely investigated and considered to be an effective approach to reduce the cost of mutation testing. In the case of large programs where source code has hundreds of classes and more than 10 KLOC lines of code, the selective mutation can still generate thousands of mutants. Executing each mutant against the test suite is cost-intensive in terms of robustness, resource usage, and computational cost. In this paper, we introduce a new approach to extract features from mutant programs based on mutant killing conditions, i.e. reachability, necessity and sufficiency along with mutant significance and test suite metrics to extract features from mutant programs. A deep learning Keras model is proposed to predict *killed* and *alive* mutants from each program. First, the features are extracted using the Eclipse JDT library and program dependency analysis. Second, preprocessing techniques such as Principal Component Analysis and Synthetic Minority Oversampling are used to reduce the high dimensionality of data and to overcome the imbalanced class problem respectively. Lastly, the deep learning model is optimized using fine-tune parameters such as dropout and dense layers, activation function, error and loss rate respectively. The proposed work is analyzed on five opensource programs from GitHub repository consisting of thousands of classes and LOC. The experimental results are appreciable in terms of effectiveness and scalable mutation testing with a slight loss of accuracy.

INDEX TERMS Scalable mutation testing, static analysis, deep learning, binary classification.

I. INTRODUCTION

Software testing plays a key role in ensuring the quality of a software system or program under test. Software testers use test suites to discover software faults when unexpected behavior is detected. Therefore, the ability of software testing to detect faults is highly correlated to the quality of test suites. To measure a test suite quality, the mutation testing identifies whether a test suite is good enough to detect those faults by making syntactic changes in a source code [1]. Alternatively, the code coverage is also considered to be an effective approach by measuring the proportion of source code executed by the test suite inputs. However, the code coverage alone does not reflect effectiveness in the test suites [2]. Mutation testing is the only promising approach to address the shortcomings of coverage-based testing [3]. In mutation testing, a significant number of faulty programs i.e. mutants are generated from the original program using mutant operators. These faulty programs are then executed against the test suites. If a test suite detects those faults, then mutants are classified as *killed*. If the test suite output is same as original program, then the mutants are classified as *alive*. The adequacy of test suites is measured using Mutation Score

The associate editor coordinating the review of this manuscript and approving it for publication was Moayad Aloqaily.

Indicator (MSI) which is a percentage of killed mutants in proportion to total number of mutants.

The scores generated by mutation testing can facilitate the software testers to locate the weaknesses in their test suites and to design new test cases accordingly. Except the assessment of a test suite quality, the mutation testing has also proven its significance in simulating realistic faults [4], localization of faults [5], and testing of the model transformations [6].

However, the cost of mutation testing is extremely high as it requires the creation and execution of each mutant with the test suite. For instance, in the study of Zhang *et al.* [7], the *Proteum* mutation testing tool for *C* programming language generates 23,847 mutants for a small program consisting of 512 lines of code. The running cost of a test suite along with 23,847 mutants can be extremely expensive. Therefore, various techniques such as weak mutation, high order mutation and selective mutation are introduced by the researchers to reduce the cost of mutation testing. The first cost reduction technique is weak mutation which identifies whether a mutant causes a state change in a program without any need to execute full program from up to the output statement [8]. The second technique is HOM testing which is generalization of at least two faults within a single mutant. The search space of HOM testing is wider compared to the traditional mutation testing and the possibility of finding interesting HOMs could reduce the proportion of generated mutants and test effort. HOM testing has been applied to model-based testing, concurrency testing as well in code-level testing [9]. The third approach is selective mutation which selects a subset of mutants to achieve a similar effect as of the whole set of mutants. In other words, if a subset of mutants is adequate to the test suite, then the effectiveness of test suite will be the same for the whole mutant set. The selective mutation has been previously studied by many researchers to find an efficient subset that represents the whole mutant set. However, the empirical study of Zhang et al. on selective mutation suggested that the existing researches are evaluated on small programs [10]. Therefore, the applicability of selective mutation is still questionable which may have scalability concerns for larger programs [7]. Even if selective mutation produces good scalability for larger programs, the efficient mutant subset still requires thousands of mutants for a program having hundreds of classes.

To address the scalability problem, we propose a new approach for mutation testing which uses a TensorFlow based deep learning Keras model to predict mutants, i.e. *killed* or *alive* without any need for test suite execution. Predictive analysis is applied by extracting features using program dependency graphs and Eclipse JDT library from each mutant. The features are extracted based on mutant killing conditions of the RIP model which states the circumstances under which a mutant should be killed.

1. ***Reachability:*** The location of the mutant is reachable by the test input. Branch or Graph coverage techniques are widely considered by researchers to detect the reachability of a mutant statement [11].
2. ***Infection:*** During test suite execution, the state of the original program is different from the mutant program which means the fault must put the mutant program into an error state [1].
3. ***Propagation:*** The infected state is propagated at some point in the mutant program after test execution such as assertions used in JUnit. [1], [12].

Other features such significance of mutant nodes are extracted based on Hyperlink Induced Topic Search (HITS) and PageRank algorithm. The distance feature is calculated by comparing the sequence of edges and nodes in dependency graphs of original and mutant programs using cosine similarity. The test suite feature *numMutantAssertions* is also extracted to measure the number of assertions covered by each mutant statement. Furthermore, the TensorFlow deep learning framework using Keras API is designed to predict the accuracy of classification results. The main contributions of this paper are summarized as follows:

1. Introducing feature extraction approaches to extract mutant features. The degree of significance, distance and test suite (coverage) features are also extracted to generate mutant datasets.
2. Mutant datasets are used to generate a binary classification model. The PCA reduction and SMOTE oversampling is used to deal with the high dimensionality of data and imbalance class problem respectively.
3. A TensorFlow based deep learning model is designed and optimized for scalable mutation testing using fine-tune parameters to predict mutations.
4. The experimental study is conducted on five open-source projects having greater than 10 KLOC from the GitHub repository to measure the scalability of the mutant classification approaches on larger programs.

The rest of the paper is organized as follows: -

Section II introduces the related work specifically on cost reduction techniques in mutation testing. Section III presents the proposed techniques for the extraction of mutant features to generate mutant datasets. The empirical study of programs, test suite details, training of models and analysis of classification results is given in Section IV. Section V introduces the possible threats to the validity experimental studies. Section VI concludes this paper and introduces the future work.

II. RELATED WORK

Mutation testing is a powerful approach to evaluate the effectiveness of test suites. The mutation testing was originally proposed by DeMillo *et al.* [3] in 1978 and since then it has increasingly gained popularity in the research community. Despite effectiveness, the mutation testing has one significant limitation which is the high cost. Since each mutant program is required to execute with the test suite to measure the testing quality. Therefore, most researches conducted in this field

are mainly focused on reducing the cost of mutation testing. The cost reduction techniques are further classified into two categories, i.e. reducing the number of mutants and reducing execution time of mutants.

A. MUTANT REDUCTION TECHNIQUES

To reduce the size of generated mutants, the selective mutation carefully selects a subset of mutants which represent the whole mutant set. Two types of selective mutation approaches are used in mutation testing, i.e. mutant operator selection and random mutant selection.

Wang and Mathor studied 22 Mothra operators and suggested that some mutant operators contribute the same as most of the mutants generated by all mutant operators [13]. An empirical study of "*sufficient mutant operators*" suggested that the mutants generated by five operators can achieve the same effectiveness as mutants generated by all mutant operators [14]. Barbosa et al. studied and proposed six guidelines for the identification of "*sufficient mutant operators*". Their study identified ten "*sufficient mutant operators*" for *C* language written programs [15]. Namin et al. suggested 28 "*sufficient mutant operators*" by combining execution information of a selected subset of mutants [16]. Gligoric et al. also studied operators and proposed "*sufficient mutant operators"* for concurrent programs [17].

Another common mutant reduction approach is random selection which randomly selects a subset of mutants from the set of all generated mutants. Wang and Mathur studied the effectiveness of the random selection technique by randomly selecting "*x%*" of mutants generated from 22 mutant operators of Mothra [18]. A study made by Zhang et al. suggested that the mutant operator selection has the same effectiveness as a random mutant selection [7]. Although the selective mutation testing has good scalability in reducing the cost of mutation testing. However, the execution of selected mutants with test suites is still cost-intensive for the larger programs.

B. TIME REDUCTION TECHNIQUES

The second category of mutation testing approaches are mainly focused on reducing the time of mutant execution.

Howden proposed the concept of weak mutation to reduce the cost of mutation testing by dividing a mutant into several components. If a state change is observed in any mutant component then the mutant is classified as *killed* [19]. Weak mutation partially executes a mutant which speeds up the mutant execution process. The weak mutation is quite faster compared to strong mutation; however, the strong mutation has more test effectiveness over weak mutation because each component produces different outputs from the original program. To overcome this issue, Woodward and Halewood proposed a compromise of weak and strong mutation known as firm mutation [20]. Firm mutation works as both weak and strong mutations by stopping program after the execution of mutant statement as well at the end of the program to save time and cost in an efficient manner [21].

Few researchers proposed parallel execution techniques to optimize the execution time of mutants. Untch et al. applied compiler manipulation to execute all mutants at once [22]. Offutt et al. applied parallel execution on several mutants to speed up the mutant execution process [23]. Zhang et al. suggested to prioritize test cases and reduce the number of test suites which could lead to the faster execution process of mutants [24]. Zhang et al. also suggested another approach that reuses the execution results of some mutants to reduce the overall execution time [25].

A recent study made by Zhang *et al.* [26] opens a new dimension in the mutation testing field. They used the ability of machine intelligence to predict mutation testing results without any need for test suite execution. Their technique is quite novel and scalable, but their feature extraction approach is highly dependent on the coverage of test suites. The coverage may cause vulnerability in the prediction of mutants if there are too many equivalent mutants. An equivalent mutant has the same semantics as the original program and produces the same behavior regardless of test suite coverage and the number of assertions used. Our feature extraction approach uses program dependence analysis of mutants to measure the RIP conditions, i.e. necessity, reachability and effective paths (paths other than the original program). Our feature extraction approach can efficiently handle this vulnerability by comparing the effective mutant paths with original program paths. If all paths are the same, then the mutant is considered as *alive* mutant regardless of the number of assertions covered the mutant statements. The detailed comparison of existing related work on cost reduction for the scalable mutation testing is shown in Table 1. Machine learning is the most promising solution to reduce the cost of mutation testing. However, machine learning may cause poor performance when used on large datasets. For instance, in the experimental study, we choose five opensource programs where train sets require thousands of mutants to train each model. Therefore, we choose the deep learning approach to make efficient predictions on large mutant datasets.

Deep learning has provided solutions to many software domains such as cybersecurity [27], [28] and the Internet of things (IOT) [29], [30]. The recent studies of Otoum et al. suggested the reinforcement and feasibility of deep learning models to make predictions on intrusion detection in software systems [31], [32]. In this paper, we designed a TensorFlow based Keras deep learning model to measure the scalability of the proposed approach on large programs. The TensorFlow is an opensource library which efficiently performs on large data in complex and heterogeneous environment. The details on feature extraction approaches and deep learning models will be explained in Section III.

III. PROPOSED APPROACH FOR FEATURE EXTRACTION USING DEEP LEARNING CLASSIFICATION

Mutation testing is rarely used in the software industry due to expensive costs. A mutant program has two alternatives, i.e. *killed* or *alive*. If a test suite detects a fault by distinguishing

TABLE 1. Comparison of existing related work on cost reduction of mutation testing.

Authors [Reference]	*Application Techniques*	*Programs Studies*	*Impact on Mutant Cost Reduction*
Wang et al. [13]	Operator Selection	Benchmark	Selection of mutant operators for Mothra
Offutt et al. [14]	Operator Selection	Benchmark	Five sufficient mutant operators
Barbosa et al [15]	Operator Selection	Benchmark	Six guidelines on 10 sufficient mutant operators for C
Namin et al. [16]	Operator Selection	Benchmark	28 Sufficient mutant operators for C
Gligoric et al. [17]	Operator Selection	Benchmark Opensource	Selection of mutant operators for concurrent programs
Wang et al. [18]	Random Selection	Benchmark	Effectiveness on random selection of N (%) mutants
Zhang et al. [7]	Random Selection, Operator Selection	Benchmark	Effectiveness on cost reduction of both random and operator selection techniques
Howden [19]	Optimizing Compilation	Theoretical Study	Weak mutation for early stopping of execution
Woodward et al. [20]	Optimizing Compilation	Theoretical Benchmark	Firm mutation for both early stopping and full test suite execution
Mayank et al. [21]	Optimizing Compilation	Theoretical Opensource	Extended firm mutation testing
Untch et al. [22]	Parallel Execution	Benchmark	Executing all mutants at the same time
Zhang et al. [24]	Test Suite Selection	Opensource	Prioritize and select a subset of test cases
Zhang et al. [25]	Reuse Execution Results	Opensource	Reuse the execution results of mutants
Zhang et al. [26]	Machine Intelligence	Opensource	Use machine learning to predict mutants

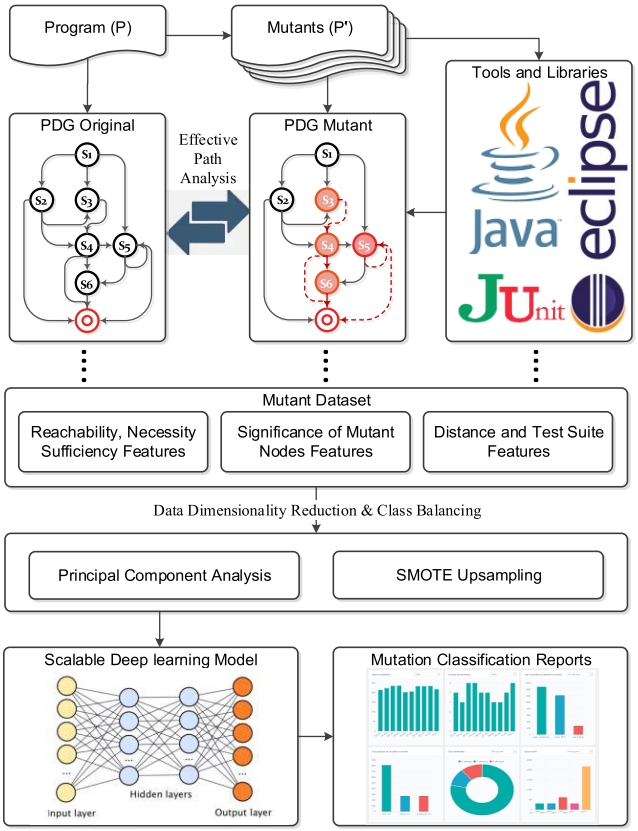

FIGURE 1. General workflow of scalable mutation testing framework.

the mutant behavior, then the mutant is classified as *killed*; otherwise mutant is classified as *alive*. Therefore, we convert the fault detection problem into a binary classification problem to use the ability of machine intelligence to predict mutants into their respective classes.

In this section, we explain how to extract features from mutant programs and how to use the deep learning models to predict mutants without any need for test suite execution. The proposed approach is divided into five steps, i.e. Mutant Generation, PDG Construction, Feature Extraction, Feature Engineering, and Deep Learning Classification. To Further demonstrate, Figure 1 is presented with the workflow of the scalable mutation testing framework using deep learning classification.

Mutant Generation: The mutation testing process starts by generating mutants from the source program. We select Mujava a mutation testing tool for Java to generate mutants. The Mujava does not support *maven*, *gradle* and *ant* build tools which are the requirement in modern java projects to build and execute program dependencies along test suites. We create a small program in Python to overcome this issue by using shell terminal commands to run mutant programs against test classes.

PDG Construction: In the second step, we use the Eclipse JDT library to generate Program Dependency Graphs from mutant programs. Eclipse JDT library provides an implementation of Abstract Syntax Trees (AST) which allows the mapping of Java source code into tree representations. The object mapping of AST is used to generate PDG for mutant programs. We compared the PDG of each mutant with the original program to identify effective paths in mutants other than original program. The detailed analysis on PDG construction is explained in Program Dependence Analysis for Extraction of Features section.

Feature Extraction In the third step, the comparison of the PDGs is used to extract four types of features from mutant programs. The reachability, necessity and sufficiency are extracted based on Constraint-Based Testing (CBT) Theory. Hub, Authority and PageRank features are extracted based on the significance of mutants. The distance feature is extracted based on semantic similarity in two programs. The test suite features are extracted based on coverage of programs by the test suites.

Feature Engineering The feature engineering is an important step of deep learning for efficient classification of results. Features are significant for predictive models and they can highly influence the prediction of results. In this paper, we use

PCA reduction to generate features with high variance from mutant programs. The total cumulative variance of 0.85 is used to select high quality features from feature sets. The SMOTE oversampling is then used to overcome imbalance class problem before the training of deep learning classifiers.

Deep Learning Classification: Deep learning has been widely studied to solve the problems of software testing. Considering the output, this paper uses a classification strategy that learns a model from some instances and then classifies new instances into different categories such as *killed* and *alive*. In this paper, the TensorFlow based Keras deep learning model is selected for scalable mutation testing and deep learning classification. We also investigate the choices of other deep learning models such as RNN and MLP to analyze their scalability with proposed features extraction approaches.

A. PROGRAM DEPENDENCE ANALYSIS FOR EXTRACTION OF FEATURES

Program Dependency Graph (PDG) is a graphical representation of program source code to visualize inner dependencies. There are two types of dependencies exist in any program such as data and control dependency. In object-oriented programming languages such as Java, C++ and C#, the program syntax consists of variables, conditions, expressions and methods calls which can be represented as nodes and edges of PDG graph. In PDG, the edges represent the call sequence and control dependencies among different statements in a program method. A PDG graph G can be characterized using four elements for each method M in Program P such that the $G = (N, E, \mu, \delta)$.

- N is a set of nodes in Method M
- $E \subseteq N \times N$ is a set of edges consist of data and control dependences among two nodes N.
- $\mu : E \rightarrow S$ defines the Node N type, i.e. variable, statement or a condition.
- $\delta : E \rightarrow T$ defines type dependency type, i.e. control or data dependency of mutant statement.

To illustrate the effects of data and control dependencies on program mutations, we use the example of a "*isTriangle*" program as shown in Figure 2.

The dependencies given in Figure 2 are represented as nodes and edges where the entry node is denoted by E and exit node is denoted by X. The solid lines indicate the presence of control dependent edge among different statements whereas the dotted lines indicate data dependent edges. The program source code shows that the Statement $S1$, $S2$, $S4$ and $S6$ are bound to be executed when the method is called by program whereas these statements only dependent on entry node E. The execution of Statement $S3$ is dependent on the execution outcome of $S2$. Therefore, $S3$ can only be executed if $S2$ is true. In PDG of "*isTriangle*" example, the control dependent edge $S2$ is a starting point and $S3$ is the ending point. Similarly, Statement $S5$ relies on $S4$ representing a control dependent edge among two statements. The statement

```
E.public bool isTriangle (float a,float b,float c)
  {
S1.     bool result = true;
S2.     if( a <= 0 || b <= 0 || c <= 0)
        {
S3.             result = false;
        }
S4.     if( a + b <= c || a + c <= b || b + c <= a)
        {
S5.             result = false;
        }
S6.     return result;
  }
```

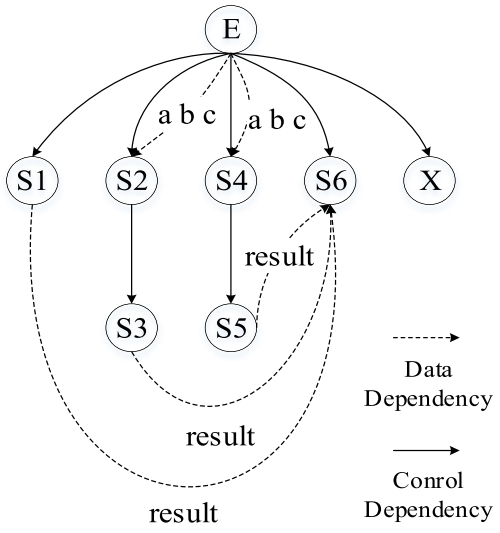

FIGURE 2. Visualization of data and control dependencies in *"isTriangle"* program using program dependency graph.

$S2$ and $S4$ uses the values of variable a, b and c passed by the program entry node. Their values are not reassigned before being referenced. In PDG of "*isTriangle*" program, there is a data dependent edge on variable a, b and c which is passed by the entry node E starting point and passes to Statement $S2$ and $S4$ as ending points. Similarly, Statement $S6$ is dependent on the value of variable *result* in Statement $S1$, $S3$ and $S5$. Using PDG analysis, the data and control dependencies can be used to identify propagation paths of mutant programs to measure the sufficiency value of mutant programs. The propagation paths are defined using three definitions given below:

Propagation Variable: A variable in a mutant statement that changes the state of PDG node.

Data Propagation Path: A starting path of a mutant node which ends at the output node, the first edge of the mutant node is data dependent on the propagation variable.

Control Propagation Path: A starting path of a mutant node which ends at the output node, the first edge of the mutant node is control dependent edge.

Abstract Syntax Tree (AST) is a powerful tool which maps the Java source code into tree representations. We used AST mapping to generate dependency graphs of the mutant programs. Three features, i.e. *reachability*,

necessity and *effectPathNum* are extracted based on propagation path analysis in mutants. We compared the propagation paths of mutant and original program to generate these features.

Since the value of each variable continuously changes during the execution of a program and the variables in each statement are used to indicate markings. Therefore, the quantitative relationship of variables and markings can be used to express the satisfiability problem. Satisfiability checks whether the reachability meets the necessity or not. We used the Microsoft Z3 SMT solver tool to measure the satisfiability problem of mutant statements. The Z3 SMT Solver is mainly composed of input statements, assertions and commands to find the satisfiability value of various logical expressions [33]. If a set of variables and their values simultaneously satisfies all assertions, then the satisfiability command returns *"sat"* otherwise Z3 SMT Solver returns *"unsat"*.

B. SEMANTIC DISTANCE FEATURE EXTRACTION

A *killed* mutant is not only syntactically but also semantically different from the original program. More the difference in semantics of a mutant from its original program, more the possibility that mutant can be killed by a test suite. Taking this theory into consideration, we extract the *distance* feature for all mutant programs. First, we use the PDG to generate edges and nodes for both mutant and original program as explained in the PDG Analysis section. Second, the dependency call sequence of nodes and edges are used to generate token strings for both original and mutant programs. Finally, the token strings or vectors are compared to measure the distance of each mutant from the original program based on cosine similarity. The detailed workflow on distance feature extraction model is given in Figure 3.

The algorithmic process for distance feature extraction starts by generating PDG graphs of both original and mutant programs where each statement $S1, S2, \ldots, Sn$ is represented as an edge of PDG such that $E = E1, E2, \ldots, X$. For instance, if Statement $S3$ is removed by Statement Deletion (SDL) operator, then it will automatically remove its corresponding Edge $E3$ from the mutant program which will also affect the call sequence of its child edges such as $E4$, $E5$, and $E6$ up to the final edge represented by the X. The call sequence of nodes starting from E3 towards other edges will be removed automatically from mutants during PDG construction such as $N3 \rightarrow N4$, then $N4 \rightarrow N6 \,\|X$ as shown in Figure 3.

Cosine similarity measures the cosine of an angle between two vectors pointing at the same direction. In our case, we used nodes and edges of PDG to measure the distinction in program before and after mutation. Therefore, the cosine similarity can efficiently measure the semantic similarity between the original and mutant program. To measure the cosine distance, let a and b are two vectors for comparison generated from PDGs where vector $a = [a_1, a_2, \ldots, a_n]$ and vector $b = [b_1, b_2, \ldots, b_n]$. The cosine similarity between

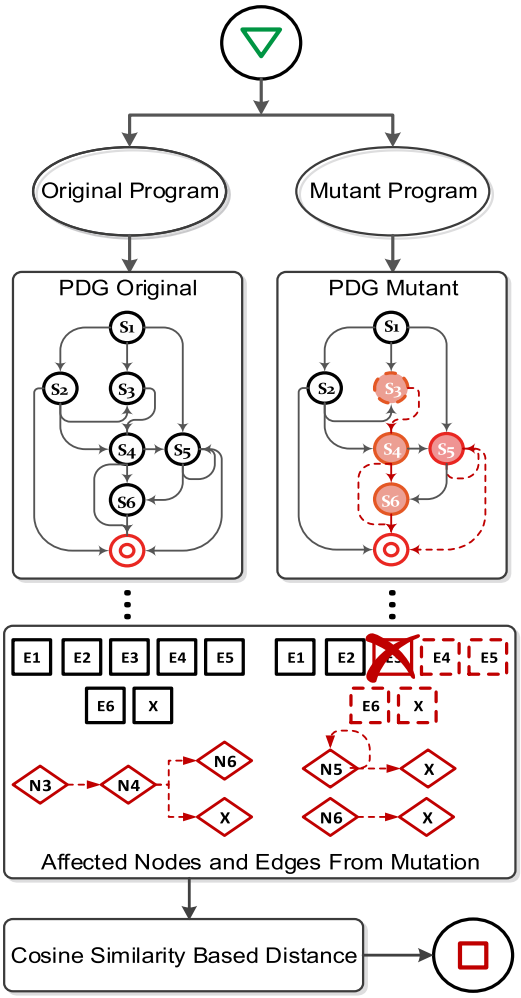

FIGURE 3. Distance feature extraction model based on program dependency analysis and cosine similarity.

a and b is calculated using Equation 1.

$$cosine\,(a,b) = \frac{\vec{a}.\vec{b}}{\|\vec{a}\|\left\|\vec{b}\right\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{1}$$

where $\vec{a}.\vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$ is dot product of two given vectors. Both vectors a and b are divided by their length to normalize them as shown in Equation 2.1 and 2.2.

$$a_{norm} = \left[\frac{a_1}{len_a}, \frac{a_2}{len_a}, \ldots, \frac{a_n}{len_a}\right] \tag{2.1}$$

$$b_{norm} = \left[\frac{b_1}{len_b}, \frac{b_2}{len_b}, \ldots, \frac{b_n}{len_b}\right] \tag{2.2}$$

From Equation 2.1 and 2.2, the normalized cosine similarity between the two vectors is measured using Equation 3.

$$cosine(a,b) = \sum_{i=1}^{n} a_{norm_i} b_{norm_i} \tag{3}$$

The cosine similarity calculates results ranging between -1 to +1. If two vectors are exactly the same then the cosine similarity is 1, and if they are exactly opposite then the cosine similarity is -1. During PDG construction, there is a possibility that the similarity of two vectors is negative. For instance, if mutant statement is the first statement and the majority of its corresponding statements are dependent on that statement, then most of the edges and nodes in the mutant vector will be affected by the mutation. Therefore, Equation 4 is used to calculate *distance* feature of each mutant also known as cosine distance.

$$\cos\theta distance = \frac{\cos^{-1} \, cosine \; similiarity}{\pi} \tag{4}$$

C. FEATURE EXTRACTION USING DEGREE OF SIGNIFICANCE

In dependence subgraph of a mutant program, each node of PDG corresponds to a statement. The control dependent edge and data dependent edge determine interrelationships between different statements from starting node E up to the ending node X. The measure of importance for a statement in program method can be transformed into the degree of significance for its corresponding nodes. It is very common to use interlinked nodes to determine the significance of a single node. However, such methods do not consider the significance of its adjacent nodes. The Hubs and Authorities also known as Hyperlink-Induced Topic Search (HITS) algorithm identifies the authoritative sources in a hyperlinked environment. The algorithm measures two scores per entity such as the hub value and the authority value of hyperlinked webpages [34]. In this study, the HITS algorithm is used to calculate the hub and the authority values of corresponding nodes for mutant statement as a measure to determine the degree of significance. Each node controls only one corresponding node and the data dependency of that node is used to determine its authority value. If more data dependent edges are pointing towards a node then more complexity it has to calculate its node expression. The hub value is determined by the data dependent edge from a node to the control dependency edge. The more edges indicate the higher influence of the current node on the other nodes. Better adequacy of a node has a higher corresponding hub value. The Algorithm 1 is presented below to measure the hub and authority value of each mutant program.

1) HITS ALGORITHM

Input: The algorithmic process starts by selecting nodes and edges of PDG as 2-tuple, i.e. {Nodes, Edges} in a 2D matrix.

- Initialize the hub and authority value of each node in PDG with 1.
- For each edge in a node, compute authority value as the sum of scaled hub values that pointed towards nodes (Equation 5).

Algorithm 1 *Check Hub and Auth value of Mutants*

Input : *Let N is set of nodes with size* $n \times m$
Output : ***Authority*** *and Hub vector*
Foreach (***node*** *in N*)
  Let $x =$ ***Hub***$[node]$
  Let $y =$ ***Auth***$[node]$
  Initialize $x = (1, 1, \ldots, 1) \in R^m$
  Initialize $y = (1, 1, \ldots, 1) \in R^n$
End Foreach
While (*N has* ***nodes***)
  For $(i = 1, 2, \ldots, m)$

$$x_j = \sum\nolimits_{a_{ij}=1} y_i \tag{5}$$

  End For
  For $(j = 1, 2, \ldots, N)$

$$y_i = \sum\nolimits_{a_{ij}=1} x_j \tag{6}$$

  End For
  Normalize (x)
  Normalize (y)
End While
Return x, y

- For each edge in a node, compute hub value as the sum of scaled authority values of a node (Equation 6).
- Each iteration consists of two steps, i.e. update authority value and update hub value.
- Finally, the algorithm returns the normalized hub and authority value as an algorithm output.

The PageRank is another popular ranking algorithm used by the Google search engine to estimate the significance of webpages [35]. In PDG, the number of nodes and edges pointed towards a single node indicate its significance on the other nodes. More influence a node has on the other significant nodes indicates its higher PageRank value on corresponding nodes. The Algorithm 2 is presented below to calculate the PageRank value of each mutant program.

2) PAGERANK ALGORITHM

Input: The algorithmic process starts by selecting nodes and edges of PDG as 2-tuple, i.e. {Nodes, Edges} in a 2D matrix

- Initialize the PageRank value of each PDG node with $\frac{1}{N}$.
- Initialize the damping factor value with 0.85 (likelihood of a random node visiting an edge)
- For each edge in a Node, identify nodes with links and without links.
- If a node has links, identify inbound and outbound links.

Algorithm 2 *Check PageRank Value of Mutants*

Input : *Let N is set of nodes with size* $n \times m$
Output : *PageRank vector*
Foreach (***node*** *in N*)
  Let PR $= PageRank[node]$
  Initialize $PR = \frac{1}{N}$
  Initialize $dampingFactor = 0.85$
End Foreach
While (*N has* ***nodes***)
  If ($node_j$ *linked to* $node_i$)
    $$hasLink = \sum_{node_j} \frac{PR(node_j)}{L(node_j)}$$
  End If
  If ($node_j$ *has no Links*)
    $$noLink = \sum_{node_j} \frac{PR(node_j)}{N}$$
  End If

$$PR\,(node_i) = \frac{1-d}{N} + d\,(hasLink + nolink) \qquad (7)$$

End While
Return *PR*

- Finally, compute the PageRank for all nodes using Equation 7 and return the cumulative PageRank value of each mutant as an algorithm output.

D. FEATURE EXTRACTION FROM TEST SUITE

The reachability, necessity and propagation features are resolved to examine the effects of machine intelligence in the prediction of mutants. At the same time, some additional features such as cosine distance and degree of significance features are also extracted to increase the efficiency of deep learning models. Besides these, some features are selected from previous studies such as [26] and [36] in which the most significant feature is *numMutantAssertion*. This feature observes the coverage of a mutant statement by the number of assertions used in the test suite. If a mutant statement is covered by at least one assertion, then there is a possibility that a mutant is killed by the test suite unless that mutant is equivalent. More the assertions have mutant coverage, more the possibility that the mutant is killed by the test suite. After the extraction of features from mutant programs, the preprocessing of data using PCA analysis and SMOTE oversampling is applied before the prediction of mutants. The detailed list of extracted features is given in Table 2.

E. SIGNIFICANCE OF EXTRACTED FEATURES

The efficient prediction of mutants into their classes, i.e. *killed* or *alive* is dependent on the significance of proposed features. In this section, we briefly explain the significance of some of these features in mutant classification.

First, we proposed three features of constraint-based testing theory, i.e. *reachability*, *necessity* and *effectPathNum*. The reachability refers to the ability of a program to be reachable by the test input. The necessity refers to the ability of mutant statements to propagate error state which should be reachable up to the output statement. Using both reachability and necessity constraints, we can calculate the *effectPathNum* value which identifies the number of paths affected by the mutations. More paths indicate more possibility of that mutant to be killed by the test suite.

Second, we proposed distance feature which measures the similarity between the semantics of the mutant and the original program. The greater distance between mutant and original program indicates a high possibility of mutant to be killed by the test suite.

Third, we proposed the significance features to generate *hub*, *authority* and *PageRank* values which indicate the interrelationship of a mutant statement to other statements within a class method. More interrelationships indicate the higher significance of a mutant statement on other statements.

Last, we proposed the coverage feature which indicates the coverage of statements in a program method. If a mutant is non-equivalent and has test suite coverage, then there is high possibility that mutant can be killed by the test suite.

TABLE 2. List of features extracted from mutant source codes to build classification models for scalable mutation testing.

No.	*Name*	*Explanation*
F01	sat	Satisfiability value of a mutant program
F02	effectPathNum	The number of propagation paths other than original program
F03	reachC	Reachability of propagation paths to the mutant statement
F04	necceC	Necessity of mutant fault to change propagation path in original program
F05	distance	Distance between semantics of mutant and original program
F06	authority	Authority value of the mutant nodes
F07	hub	Hub value of mutant nodes
F08	pagerank	Significance of mutant node on other nodes in a method
F09	numMutantAssertion	Number of assertions in a test suite cover the mutant statement
F10	typeOperator	The mutant operator used to insert sytactic fault
F11	typeStatement	The type statement of mutated method
F12	typeReturn	The type return of mutated method
F13	PAR	The number of parameters used in mutated method
F14	LOC	Number of valid lines in mutated method
F15	depNestBlock	Basic block nesting level in mutated method
F16	class	The target class for prediction i.e. *killed* or *alive*

Moreover, we also selected some other state of the art features from literature study shown in Table 2 to facilitate the deep learning models for prediction of mutants into *killed* and *alive* classes without any need of real time execution of test suites on mutant programs.

F. PREPROCESSING OF DATASETS

The feature extraction from mutant programs is a key step to generate mutant datasets for prediction. However, datasets alone do not have enough potential to produce scalable results from deep learning models. For instance, each opensource program used in this study has more than 10,000 mutants where each mutant consists of 15 features and collectively one mutant dataset has more than 1,50,000 features. In mutation testing, the independent dataset has more effectiveness compared to the data split ratios. Therefore, during the training of deep learning classifiers all programs except one are used to train models. The remaining program is used as a test set for prediction. Currently, we are confronting four challenges in our datasets needed to be resolved before training of deep learning classifiers.

1. ***Data dimensionality:*** In the feature extraction model, we extract both nominal and categorical features from mutant programs. At the same time, each training model consists of more than 40,000 mutants. The high dimensionality of data makes it difficult for training models to predict certain quantities.
2. ***Multicollinearity:*** If a dataset has a perfectly positive or negative correlation between its attributes, then there is a high possibility that classification models are affected by multicollinearity. A multicollinear data may produce skewed or misleading results during classification. In the feature extraction model, the *effectivePathNum* feature has a positive correlation with the *reachability* and *necessity* features. If *reachability* and *necessity* value increases, then the value of *effectivePathNum* will also increase. The *reachability* and the *necessity* feature have a moderate correlation. For instance, a *necessity* of mutant can only be measured if the mutant statement is reachable. If *reachability* is zero or negative, then the *necessity* of mutant may also be zero or negative too. Similarly, the *authority* and *hub* values are extracted based on correlation to each other.
3. ***Train and Test Set Compatibility:*** The splitting methods provided by *weka* or *sklearn,* etc. splits data ratios into compatible train and test sets. However, in case of independent test set, features are independently extracted from each opensource program. Therefore, using one program as test set and other programs as train set may cause compatibility problem during classification. For instance, depNestBlock, *typeStatement* and *typeReturn* features can be different in each program. If a *typeReturn* of a method is a class name, then mutants generated on that method cannot be predicted by the independent train set.
4. ***Imbalance Classes:*** In mutation testing, the number of *killed* and *alive* mutants is highly dependent on the quality of the test suites. For instance, if a test suite is adequate, then the majority of mutants will be killed by the test suite. Similarity, if a test suite has weak adequacy or low coverage, then only a few mutants can be killed by the test suite.

1) PRINCIPAL COMPONENT ANALYSIS (PCA)

To resolve the challenges of the high dimensionality of data and multicollinearity, we performed the principal component analysis on mutant datasets. The PCA is a statistical approach which converts the highly correlated features of data into uncorrelated variables known as Principal components i.e. $PC_1, PC_2, \ldots, PC_n$. The PCA uses orthogonal transformation on high dimensional data by reducing it into low dimensional space without any loss of actual information [37]. The PCs generated by the PCA algorithm maps the high variance of all datasets into eigenvalues. Several PCs can be generated from data; however, the PCs are generated in such a way that the first PC contains the highest variance than second and so on. This approach automatically removes the noisy data and chooses uncorrelated and complex data to generate PCs which simplifies prediction analysis for deep learning classifiers. The new features generated by PCs reflect all the information contained in the original datasets [38]. After extraction of PCs from each dataset, those PCs which collectively contain more than 80% up to 90% variance are selected for further analysis. The covariance among two variables shows the importance of specific PC in deep learning prediction whereas each PC value ranges between +1 to -1. The covariance between two random dimensions A and B from n number of dimensions are calculated using Equation 8.

$$cov\,(A, B) = \frac{\sum_{n}^{i=1} (A_i - \overline{A})\left(B_i - \overline{B}\right)}{n-1} \tag{8}$$

Mathematically, the p dimensional vector with individual coefficient weights are expressed using Equation 9.

$$w_k = (w_1, w_2, \ldots, w_p)_k \tag{9}$$

The w_k is a unit vector with k linear number. Each feature in vector is transformed into linear form to capture variance t and use variance to compute the weight as shown in Equation 10 and 11.

$$t_i = (t_1, t_2, \ldots, t_n)_i \tag{10}$$

$$t_{k(i)} = x_i.w_k \tag{11}$$

The x represents the maximum possible variance along with coefficient vector w for each variance. The first PC with highest variance is calculated using Equation 12. Further, each successive PC is used to captures the next highest variance.

$$w_1 = arg_{\|w\|=1} max \left\{ \sum_i (X \times w)^2 \right\} \tag{12}$$

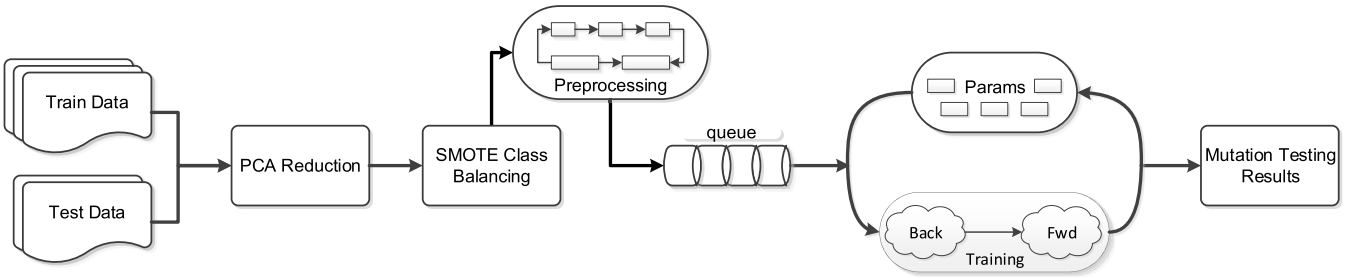

FIGURE 4. TensorFlow dataflow graph for preprocessing, training pipeline and mutation testing classification.

The w is weight for the first principal component and X is a data matrix. To find the k^{th} PC, we subtract the first principal component $k-1$ from X.

$$\hat{X_k} = X - \sum_{s=1}^{k-1} X \times w_s \times w_s^T \tag{13}$$

To calculate the weight vector to extract the k^{th} maximum variance of next PC, the Equation 14 is presented.

$$w_k = arg_{\|w\|=1} max \left\{ \left\| \hat{X_k} \times w \right\|^2 \right\} \tag{14}$$

The high dimensionality of data and multicollinearity issues are resolved using PCA reduction and PCs selection techniques. To make train and test data compatible with each other, we independently perform PCA on each train and test set. The PCs generated from PCA usually have a positive or negative numeric value. We choose an equal number of PCs in both train $(PC_1, PC_2, \ldots, PC_n)$ and test $(PC_1, PC_2, \ldots, PC_n)$ sets to make them compatible before making predictions.

2) IMBALANCE DATA ISSUE

In mutation testing, the proportion of *killed* and *alive* mutants may vary greatly depending on the quality of test suites. In such scenarios, the overfitting or imbalanced class problem arises which may affect the classification of mutants. Various approaches have been proposed by the researchers to overcome imbalance data problems. In this study, we select Synthetic Minority Oversampling (SMOTE) approach which has been previously used by many researchers because it significantly improves the accuracy of minor classes [39]. The SMOTE performs a k-nearest neighbor approach to generate syntactic samples of minor classes using the following steps.

- Calculate k-nearest neighbor for each entity of minor class using Euclidean distance such that $x_i \in S_{min}$.
- Select a random nearest neighbor x_j from the group of k-nearest neighbor x_i.
- The new sample will be generated based on 3 or 5 nearest neighbors using Equation 15.

$$x_{new} = x_i + \left| x_i + x_j \right| \times \delta \tag{15}$$

where $\delta \in [0, 1]$ is a random number between 0 and 1 which is used for the placement of newly generated samples. Deep learning models usually encounter this problem during the training of classifiers when one class dominates the other. The SMOTE oversampling on classification models is used to overcome this problem.

G. DEEP LEARNING WITH TENSORFLOW FRAMEWORK

We design our main deep learning model using the TensorFlow library based on Keras API to predict mutants from unseen test data.

TensorFlow Keras Significance: We choose TensorFlow because it efficiently works on any data in complex and heterogeneous environments. TensorFlow allows high level computation, training of models, tracking and sharing of operations to mutate dataflow graphs. It provides a flexible environment for an application developer to design and optimize deep learning models using hyperparameters. TensorFlow uses multi-dimensional arrays to perform operations known as tensors. The queue feature in the TensorFlow performs parallel execution similar to the multi-threading to speed up operations used for classification [40]. Keras is a user-friendly API that enables fast prototyping and provides configurability for new modular extensions [41]. Keras runs smoothly on both CPU and GPU. Keras supports almost all neural network models such as fully connected, sequential, convolutional, recurrent and embedding, etc. Furthermore, the flexible and modular nature of Keras model enables users to combine multiple models to build more complex deep learning models.

The proposed feature extraction and dimension reduction techniques are designed to work on the proposed deep learning model. However, their application is not limited to the TensorFlow but can also work on other types of deep learning models. Therefore, in this study, we also select some other models such as RNN and MLP to measure the flexibility and scalability of mutation testing prediction. The detailed workflow of the TensorFlow based Keras model and the steps used in the preprocessing of data are shown in Figure 4.

1) MODEL DESIGN

The backward and forward process with fine-tuned parameters is used to train deep learning models. The fine-tune procedure is optimized with different parameters such as activation and loss functions, dropout layers, optimizer and

learning rate. We configured seven layers to train features where the first six layers' neuron sizes are 120, 100, 80, 60, 6 and 40 respectively. The first layer in the deep learning model is used as an input layer whereas the last layer is an output layer. In the middle, five hidden layers are assigned where artificial neurons use a set of weighted inputs to learn the model and produce output using an activation function. The activation function *sigmoid* is used with the output layer which represents the nonlinear form of neural network. The main advantage of the *sigmoid* is that the output values exist between 0 and 1 only. Therefore, in those models where the prediction is dependent on the probability of occurrence, the *sigmoid* performs better. For instance, we extract features based on mutant killing conditions. If a mutant satisfies these conditions, then the probability of mutant is *killed* higher than the probability of mutant is *alive*. The neural network measures the linear arrangement of input signals and uses *sigmoid* function to deliver the outcome in the fixed output range. *Sigmoid* is a standard logistic function which can mathematically be defined using Equation 16.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{16}$$

where S is sigmoid function and e is an exponential function of variable x. In deep learning classification models, the combining predictions from multiple neural networks can overfit the model whereas the dropout layer randomly drops some units during the training of neural networks to prevent too much co-adoption [42]. To overcome the overfitting problem, a dropout layer with the dropout ratio of 0.20 is used with each hidden layer in the deep learning model.

2) MODEL TRAINING AND EVALUATION

The main goal of training a model is to learn from the structure of data and make predictions on the unknown test set. The optimizer and loss functions can contribute to improve the training of the designed model. In this study, we use *Adam* optimizer which is the combination of adaptive gradient and root mean square propagation algorithm making it a computationally efficient optimizer. *Adam* performs an iterative approach to update neural network weight for each parameter in the deep learning network [43]. *Adam* stores the exponential decaying average of the past square gradient v_t and the decaying average of the past gradient m_t in similar momentum. We compute both gradients using Equation 17 and 18 respectively.

$$m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times g_t \tag{17}$$

$$v_t = \beta_2 \times m_{t-1} + (1 - \beta_2) \times {g_t}^2 \tag{18}$$

The g identifies the particular gradient for every moment, i.e. the mean or the uncentered variance of a gradient. To counteract biases, the corrected first $\hat{m}_t$ and second $\hat{v}_t$ moment is estimated using Equation 19 and 20.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{19}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{20}$$

Lastly, we select the binary cross entropy loss function to measure the loss in classification results. The proposed mutant classification model consists of only two target classes, i.e. *killed* or *alive*. Therefore, binary cross entropy is more appropriate for our datasets which efficiently calculates the difference between two distributions. Mathematically, the binary cross entropy is defined using Equation 21.

$$BCE(t, 0) = -(t \times \log(o) + (1 - t) \times \log(1 - o)) \tag{21}$$

where t is the target and o is an output of the symbolic TensorFlow.

H. CLASSIFICATION WITH RNN AND MLP

The second and third deep learning model selected for the evaluation of proposed approaches is the Recurrent Neural Network (RNN) and Multi-Layer Perceptron (MLP) model respectively.

RNN is a form of artificial neural network in which previous states are used as input for the current states. The structure of RNN is similar to the Feedforward Neural Network with one distinction in the usage of network states. To implement the RNN model, we use the same configuration of hyperparameters used in the TensorFlow Keras Deep learning model. However, the activation function *ReLU* is considered instead of *sigmoid* for better performance.

MLP is a simple form of Feedforward Neural Network which consists of three layers, i.e. input, hidden and output layer. MLP uses nonlinear activation function by default. We selected the AutoMLP classifier which automatically configures hyperparameters whereas the size of training cycles 20 and the size of ensembled MLPs 5 is used to design the deep learning model.

I. ADEQUACY METRICS FOR MODEL EVALUATION

We assessed the prediction of deep learning classification results using *True Positive (TS)*, *True Negative (TN)*, *False Positive (FP)* and *False Negative (FN)* variables respectively. The evaluation metrics used for analysis are presented in Equation 22, 23, 24 and 25 respectively.

$$Precision = \frac{TP}{TP + FP} \tag{22}$$

$$Recall = \frac{FP}{FP + TN} \tag{23}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{24}$$

$$F_{measure} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{25}$$

IV. EXPERIMENTAL STUDY AND ANALYSIS

Experiments are designed to test the effectiveness and scalability of proposed techniques under different application scenarios such as program size, accuracy and loss. We select five programs for experimental study. Firstly, the test suites

TABLE 3. Information of five opensource Java programs selected for experimental study.

# Subject Programs	Method LOC	Test suite LOC	#Classes and #Test Cases	Program Coverage
mixprojs	*10,432*	*07,371*	*152/664*	*85.17*
comcodec	*12,153*	*15,819*	*179/490*	*96.80*
linq4j	*13,767*	*08,542*	*218/632*	*71.20*
comtext	*21,817*	*11,994*	*240/1031*	*87.80*
jfreechart	*36,141*	*10,858*	*653/2355*	*73.50*
Total	*94,310*	*54,584*	*1,498/5,172*	*82.89*

NOTE: *The details of programs are disclosed in the following GitHub repository link: https://github.com/deepmut/programs*

TABLE 4. Mutant operators used in experimental study.

Mutant Operator	Description
AOR	*Arithmetic Operator Replacement*
	Example: $[a + b \Rightarrow a - b]$
AOI	*Arithmetic Operator Insertion*
	Example: $[a + b = c \Rightarrow a + b - -= c]$
COR	*Conditional Operator Replacement*
	Example: $[if(a + b \;\&\&\; true) \Rightarrow if(a + b \;||\; true)]$
COI	*Conditional Operator Insertion*
	Example: $[if(a > b) \Rightarrow if(!(a > b))]$
LOR	*Logical Operator Replacement*
	Example: $[a = b \;\&\; c \Rightarrow a = b \mid c]$
LOI	*Logical Operator Insertion*
	Example: $[a = b \Rightarrow a = \sim b]$
ROR	*Relational Operator Replacement*
	Example: $[if(a == 0 \Rightarrow if(a < 0)]$
SDL	*Statement Deletion*
	Example: $[if(true)\{a = b;\} \Rightarrow if(true)\{\}]$

are executed against mutants to establish the ground truth of *killed* and *alive* mutants. Secondly, the predictions are utilized on test data to answer the following questions.

RQ1: How does the proposed approach perform in terms of predicting mutants on unseen data?

RQ2: How do different factors, i.e. size, accuracy and loss etc. influence the scalability of deep learning models?

A. SUBJECT PROGRAMS

We choose five subject programs from the GitHub repository to evaluate our proposed techniques on opensource projects. MuJava mutation testing tool is used to generate mutant versions of original programs. Researchers have widely used MuJava for mutation testing of Java programs. MuJava generates both source and compiled files of each mutant which makes it easier to perform static analysis on mutant programs' source code for feature extraction. The compiled files optimize the test suite execution without any need for precompilation especially in large programs. The detailed information about subject programs such as name, method lines of code (LOC), test suite LOC, number of classes, number of test cases and test suite coverage for each program is given in Table 3. The size each program ranges between 10,432 to 36,141 LOC whereas the LOC of test suites is ranging between 7,371 to 15,819 respectively. It should be noted that the information detail of each program is extracted using eclipse metric tool. The test suite detail in some programs also includes test suites generated using Evosuite test generation tool. Evosuite uses mutation testing to generate test oracles to ensure higher structural coverage with a minimal number of test cases [44].

B. MUTANT OPERATORS AND MUTANT GENERATION

In mutation testing, the artificial syntactic changes are seeded in original program to generated its mutant versions. These syntactic changes are induced by manipulating an arithmetic operator or conditional operator, etc. in a program source code. In this study, we selected eight traditional operators of Mujava to generate mutants from subject programs. The mutant operators of Mujava are selected based on "*Sufficient Mutant Operators*" criteria studied in the Related Work section and Table 1.

We select five types of sufficient mutant operators, i.e. *Arithmetic Operator*, *Conditional Operator*, *Logical Operator*, *Relational Operator* and *Statement Deletion*.

- An arithmetic mutation is inserted by replacing the "+" with "−" or "/" with "∗" etc. in a source code statement or expression.
- A conditional mutation is inserted by replacing "||" with "&&" etc. in a source code statement or expression.
- A logical mutation is inserted by replacing "|" with "&" or adding "∼" before the variable in a source code statement or expression.
- A relational mutation is inserted by replacing "==" to "<=", ">=" or "! =" in a program statement or expression.
- A statement deletion mutation is inserted by removing a statement or an expression in a source code of program method.

The detailed list of mutant operators and examples is given in Table 4. After mutant generation, each mutant is executed against test suites to identify *killed* and *alive* mutants. The detailed information on test cases and programs is given in Table 3. In Java, the test oracles use JUnit assertions to identify program failures. A failure is an inconsistency between expected and delivered the output of a program under test. The execution results of test suites are used to calculate *MSI* scores for each program. In this paper, we use original test suites written by the software developers and

TABLE 5. The detailed information on total number of mutants and MSI scores based on outcomes of test suite execution.

# Subject Programs	Total Mutants	Killed Mutants	Alive Mutants	Mutation Score Index (MSI)
mixprojs	*10,431*	*6,634*	*3,797*	*63.59*
comcodec	*10,297*	*8,590*	*1,707*	*83.42*
linq4j	*10,891*	*4,753*	*6,138*	*43.64*
comtext	*11,750*	*9,175*	*2,575*	*78.08*
jfreechart	*13,354*	*6,042*	*7,312*	*45.24*
Total	*56,723*	*35,194*	*21,529*	*62.02*

TABLE 6. Train and test set distribution for all project based on number of mutants and number of training features.

Sr.	Train Mutants	Train Features	Test Set
1.	*46,292*	*740,672*	*mixprojs*
2.	*46,426*	*742,816*	*comcodec*
3.	*45,832*	*733,312*	*linq4j*
4.	*44,973*	*719,568*	*comtext*
5.	*43,369*	*693,904*	*jfreechart*

the contributors of the subject programs. In two programs, i.e. *mixprojs* and *linq4j*, the coverage of source code by the original test suites was less than 50%. Therefore, we use Evosuite tool to generate more test cases for some classes to further improve the quality of test suites. The detailed list on total number of mutants and *MSI* for all programs is given in Table 5. The total number of mutants generated from selected classes is 56,723. The total number of mutants killed by the test suites are 35,194 whereas the total number of alive mutants are 21,529 respectively. The highest *MSI* score is achieved by the *comcodec* program which is 83.42 whereas the lowest *MSI* score is 43.64 achieved by the *linq4j* program.

C. TRAIN AND TEST SETS DISTRIBUTION FOR CLASSIFICATION OF MUTANTS

In this paper, we choose the *cross-project* scenario for classification of mutant programs into their *alive* and *killed* classes respectively. In the *cross-project*, the train and the test models are selected independently without any split ratio criterion. Each independent train set consists of mutants from four subject programs whereas the mutants in the remaining program are used as a test set to make predictions. Train sets used in this study consist of more than 40,000 mutants whereas total features in each train model are ranging between 693,904 and 742,816 respectively. The details on number of mutants in training models, total features in each train set and programs used as test set are shown in Table 6. In some programs where MSI scores are too high or too low, the imbalance data strategy explained in section III is used to prevent deep learning models from overfitting. Lastly, we used the adequacy metrics such as *precision*, *recall* and *f-measure* to evaluate the performance of deep learning models.

D. PCA REDUCTION ON TRAIN AND TEST MODELS

The PCA reduction is used on both train and test models to reduce the number of features and to select features with high variance. This will not only reduce the dimensionality of data but also improve the ability of deep learning models to classify mutants into *killed* and *alive* classes. Firstly, we used *NominalToNumeric* conversion on data to convert categorical variables into numeric data. The resultant datasets are used to check correlation among different features. The correlation analysis of the combined dataset is shown in Figure 5. It can be seen that the majority of variables are nonlinear and highly correlated to each other. This highest positive correlation lies between *reachability* and *LOC* features whereas the highest negative correlation lies between *necessity* and *PAR* features which is 0.72 and -0.59 respectively. In the next step, we applied principal component analysis on a combined dataset to visualize the correlation of principal components generated from mutant features as shown in Figure 6. The visualization of resultant dataset shows only numeric values of data whereas the Principal Components (PCs) generated from PCA shows high variance and zero correlation among the different combination of principal components.

Cumulative Variance: The cumulative variance gives the percentage of variance covered from the first to the *n* number of principal components. The recommended and widely used selection criteria for cumulative variance is 0.85. The total cumulative variance of 0.85 means that up to that ratio the 85% variability of data is already covered by 1 to the *n* number of principal components. Therefore, the remaining principal components are considered noisy and can be removed from data without any loss of useful information.

After generating principal components for each program, the noisy data is removed based on total cumulative variance. The principal components capturing high quality features into lower dimensional space are selected. The total number of fifteen PCs are generated per mutant for all subject programs. The standard deviation, proportion of variance and cumulative variance of each PC in each program is presented in Table 7. In three programs *mixprojs*, *comtext* and *jfreechart,* the 85% cumulative variance is covered by the first nine PCs. The 0.85 cumulative variance of *comcodec* is covered by the first ten and the cumulative variance of *linq4j* is covered by the first eight PCs respectively. However, during the training of deep learning models, we require an equal number of PCs for all programs. Therefore, we select the average cumulative variance of five programs which is covered by the first nine PCs. The proportion of variance from PC10 to PC15 is very small whereas the 85% cumulative variance is already covered in PC1 to PC9. Consequently, PC10, PC11, PC12, PC13, PC14 and PC15 are considered noisy and removed from datasets.

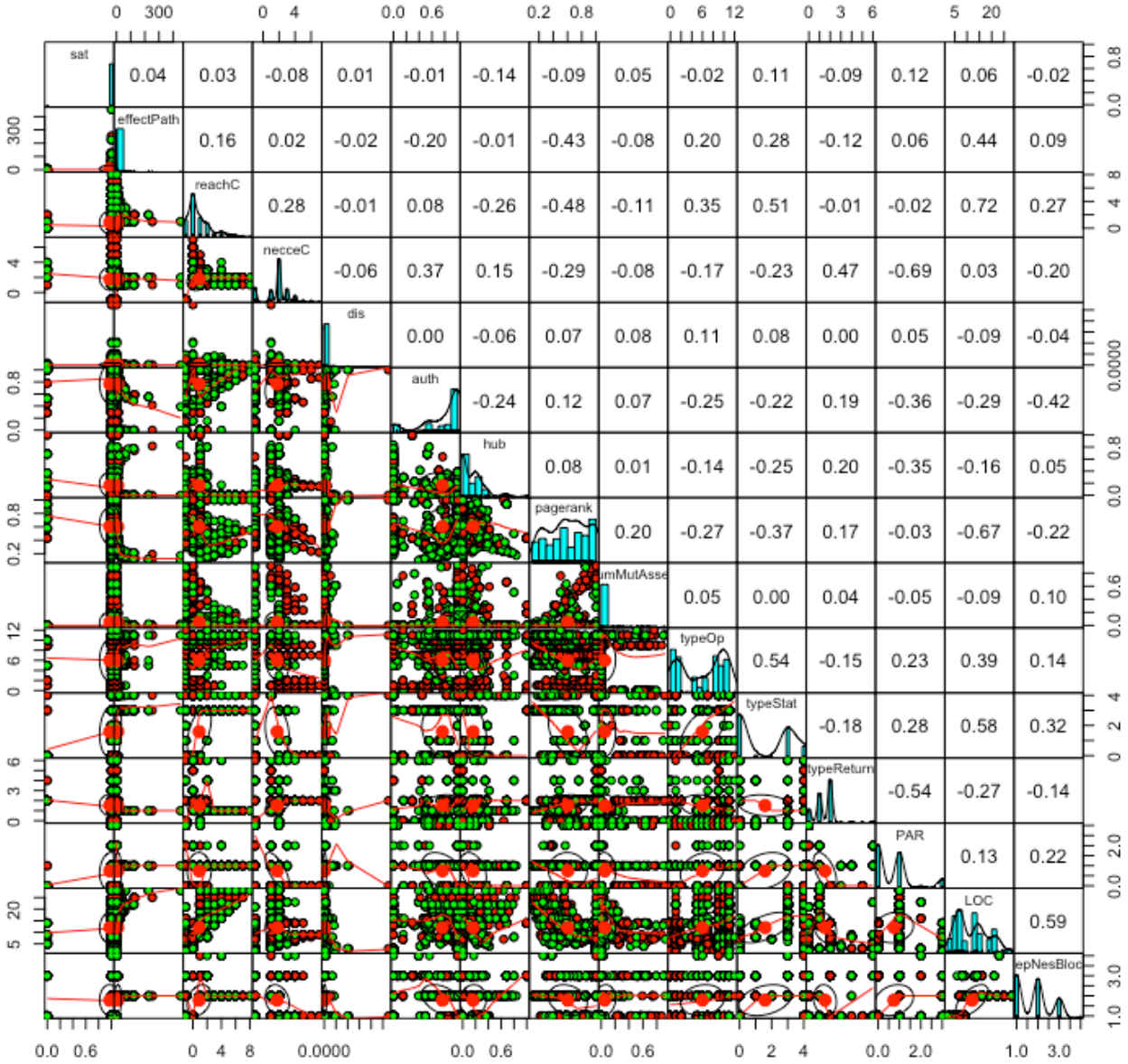

FIGURE 5. Non-linear and highly correlated values in datasets.

E. PERFORMANCE OF DEEP LEARNING MODELS IN CLASSIFICATION OF MUTANTS

1) PERFORMANCE ON ADEQUACY OF MODELS

The performance analysis of deep learning models in classification of mutants is shown in Figure 7, 8 and Table 8. We also select the choice of other deep learning classifiers such as Recurrent Neural network (RNN) and Multi-Layer Perception (MLP) to analyze the efficiency and scalability of proposed approaches with other deep learning models. Figure 7 shows the accuracy comparison of five subject programs based on three deep learning models. From the comparison of results, we can conclude that the deep learning Keras model outperforms other classifiers in all programs. The lowest accuracy of the proposed model is 0.87 whereas the highest accuracy is 0.93 which shows good effectiveness of mutant classification techniques. The RNN model also shows good classification results by achieving classification accuracy between 0.77 and 0.90 respectively. MLP model achieves the lowest accuracy of 0.73 in *comcodec* program which is still effective in terms of reducing mutation testing cost especially for large programs used in this study.

We also select other adequacy metrics such as *precision*, *recall* and *f-measure* to measure the performance of proposed techniques on deep learning models. The comparison of *precision* and *recall* is shown in Figure 8. It should be noted that in mutation testing, the imbalanced class problem is a common issue. Therefore, we analyze the *precision* and *recall* of *alive* and *killed* mutants separately. For instance,

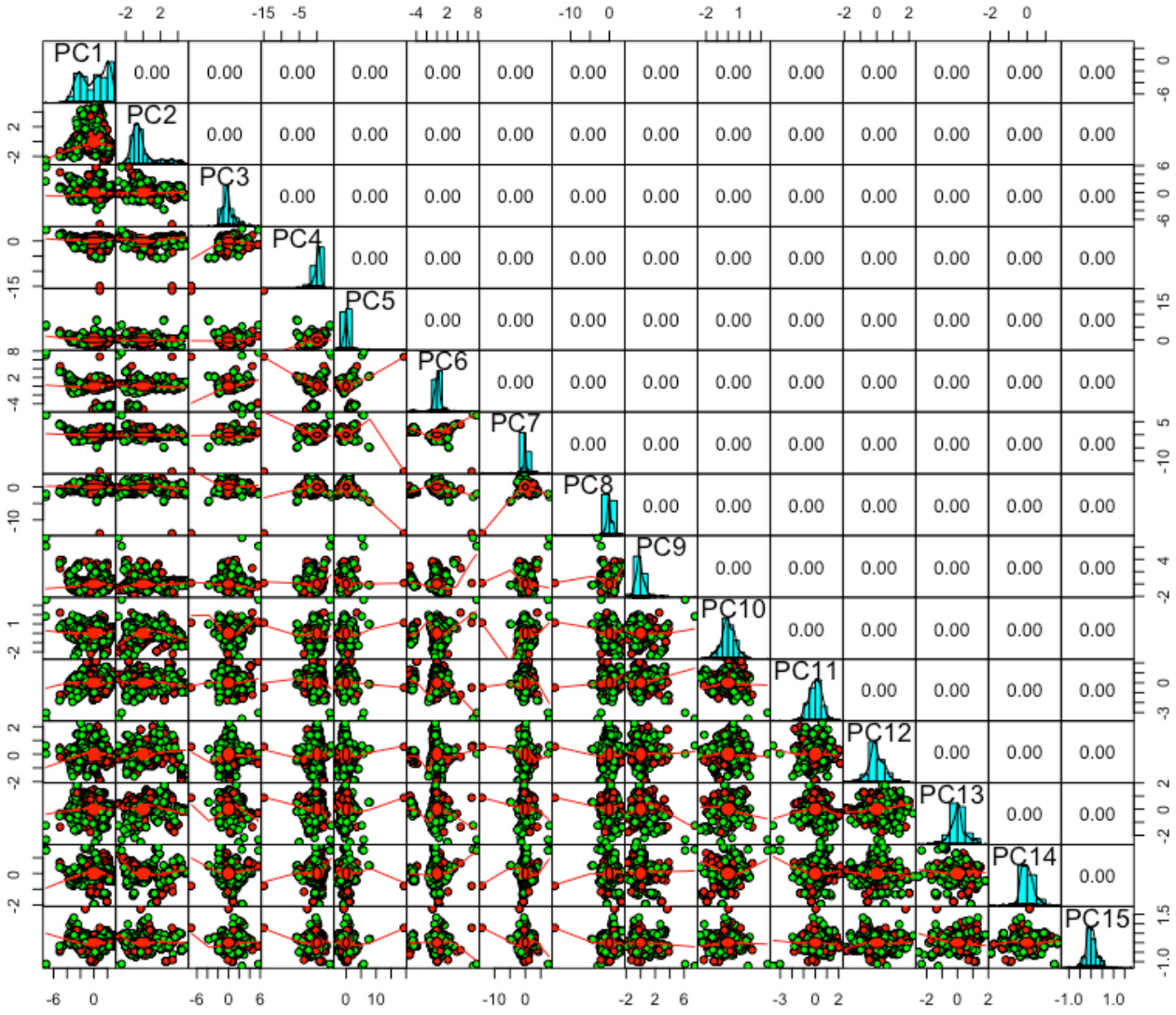

FIGURE 6. Linear and non-correlated Principal components (PCs).

using TensorFlow based Keras deep learning model, the average *precision* and *recall* of alive mutants for all programs is 0.91 whereas the average *precision* and *recall* of killed mutants is 0.86 and 0.85 respectively. The lowest *precision* and *recall* using this model is achieved by *linq4j* which is 0.88 and 0.89 respectively. One possible reason for that is the mutation score of *linq4j* is low which means the majority of mutants in this program are alive. However, the *precision* and *recall* of *linq4j* is still greater than 0.86 for both *alive* and *killed* mutant classes. The performance on deep learning classification showed good tradeoff between efficiency and effectiveness even for the highly imbalanced programs.

2) PERFORMANCE ON SCALABILITY OF MUTATION TESTING

The deep learning models are tended to overfit which means a training model is too well that it can negatively impact the performance on test models regardless of good classification results. To overcome this bias, the regularization or dropout layers are used to design deep learning models as explained in Deep Learning with TensorFlow section. To measure the scalability of mutation testing on a large test sets, a new experiment is designed to visualize accuracy and loss of both train and test models on multiple epochs. In this experiment, three programs *comcodec*, *linq4j* and *jfreechart* are used as train set whereas *mixprojs* and *comtext* are used as test set. The train model consists of 34,542 mutants or 518,130 features whereas the test model consists of 22,181 mutants or 332,715 features respectively. After training of the deep learning model with train data, the predictions are performed on the test data. The *precision* and *recall* of *alive* classes are 0.96 and 0.97 whereas the *precision* and *recall* of *killed* classes are 0.95 and 0.93 respectively. The overall accuracy of test model is 0.96 which shows good scalability of training models not

TABLE 7. The standard deviation, proportion of variance in each PC and cumulative variance for five subject programs.

mixprojs	*PC1*	*PC2*	*PC3*	*PC4*	*PC5*	*PC6*	*PC7*	*PC8*	*PC9*	*PC10*	*PC11*	*PC12*	*PC13*	*PC14*	*PC15*
Std. Dev.	2.042	1.501	1.157	1.072	1.012	0.998	0.964	0.845	0.835	0.709	0.666	0.573	0.503	0.344	0.264
Prop. Ver.	0.278	0.150	0.089	0.076	0.068	0.066	0.061	0.047	0.046	0.033	0.029	0.021	0.016	0.007	0.004
Cum. Ver.	0.278	0.428	0.517	0.594	0.662	0.729	0.791	0.838	**0.885**	0.918	0.948	0.970	0.987	0.995	1.000
comcodec	***PC1***	***PC2***	***PC3***	***PC4***	***PC5***	***PC6***	***PC7***	***PC8***	***PC9***	***PC10***	***PC11***	***PC12***	***PC13***	***PC14***	***PC15***
Std. Dev.	1.627	1.431	1.277	1.091	1.011	1.003	0.962	0.933	0.909	0.866	0.803	0.730	0.639	0.561	0.410
Prop. Ver.	0.176	0.136	0.108	0.079	0.068	0.067	0.061	0.058	0.055	0.050	0.043	0.035	0.027	0.021	0.011
Cum. Ver.	0.176	0.313	0.422	0.501	0.569	0.636	0.698	0.756	0.811	**0.861**	0.904	0.940	0.967	0.988	1.000
linq4j	***PC1***	***PC2***	***PC3***	***PC4***	***PC5***	***PC6***	***PC7***	***PC8***	***PC9***	***PC10***	***PC11***	***PC12***	***PC13***	***PC14***	***PC15***
Std. Dev.	1.746	1.622	1.458	1.210	1.010	0.999	0.944	0.919	0.690	0.627	0.614	0.520	0.478	0.384	0.266
Prop. Ver.	0.203	0.175	0.141	0.097	0.068	0.066	0.059	0.056	0.031	0.026	0.025	0.018	0.015	0.009	0.004
Cum. Ver.	0.203	0.378	0.520	0.618	0.686	0.753	0.812	**0.868**	0.900	0.926	0.952	0.970	0.985	0.995	1.000
comtext	***PC1***	***PC2***	***PC3***	***PC4***	***PC5***	***PC6***	***PC7***	***PC8***	***PC9***	***PC10***	***PC11***	***PC12***	***PC13***	***PC14***	***PC15***
Std. Dev.	1.723	1.533	1.212	1.096	1.026	1.000	0.971	0.932	0.878	0.861	0.759	0.676	0.591	0.433	0.232
Prop. Ver.	0.197	0.156	0.098	0.080	0.070	0.066	0.062	0.058	0.051	0.049	0.038	0.030	0.023	0.012	0.003
Cum. Ver.	0.197	0.354	0.452	0.532	0.603	0.669	0.732	0.790	**0.842**	0.891	0.930	0.960	0.983	0.996	1.000
jfreechart	***PC1***	***PC2***	***PC3***	***PC4***	***PC5***	***PC6***	***PC7***	***PC8***	***PC9***	***PC10***	***PC11***	***PC12***	***PC13***	***PC14***	***PC15***
Std. Dev.	1.762	1.437	1.318	1.135	1.089	0.999	0.971	0.922	0.896	0.782	0.712	0.655	0.600	0.258	0.207
Prop. Ver.	0.207	0.137	0.115	0.085	0.079	0.066	0.062	0.056	0.053	0.040	0.033	0.028	0.024	0.004	0.002
Cum. Ver.	0.207	0.344	0.460	0.546	0.625	0.692	0.754	0.811	**0.865**	0.906	0.939	0.968	0.992	0.997	1.000

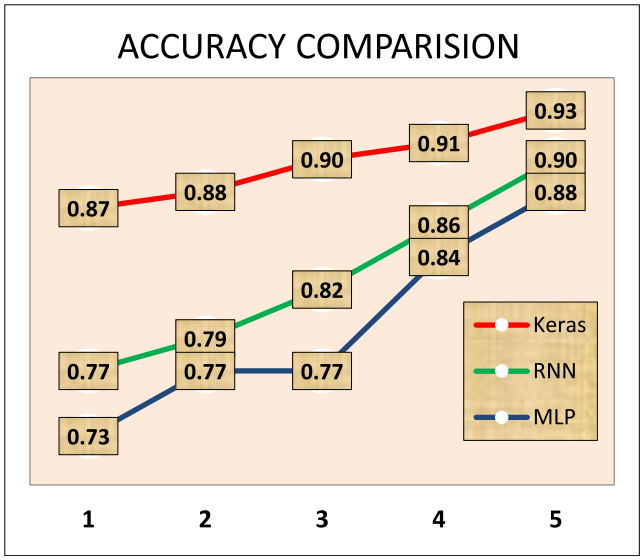

FIGURE 7. Accuracy comparison of TensorFlow Keras, RNN and MLP models on five subject programs.

TABLE 8. The performance analysis of mutant classification into alive and killed classes based on three deep learning models i.e. proposed TensorFlow Keras model, RNN and MLP.

Classifier	*# Subject Programs*	*Alive Mut's*		*Killed Mut's*		$F_{measure}$
		Prec.	*Reca.*	*Prec.*	*Reca.*	
Proposed Keras Deep Learning Model	*mixprojs*	0.91	0.95	0.91	0.84	0.90
	comcodec	0.94	0.97	0.89	0.80	0.90
	linq4j	0.88	0.89	0.87	0.86	0.87
	comtext	0.94	0.92	0.81	0.86	0.88
	jfreechart	0.90	0.85	0.85	0.90	0.86
	Average	0.91	0.91	0.86	0.85	0.88
Recurrent Neural Network Model	*mixprojs*	0.83	0.76	0.87	0.91	0.79
	comcodec	0.71	0.88	0.86	0.67	0.78
	linq4j	0.86	0.82	0.88	0.92	0.88
	comtext	0.72	0.91	0.90	0.70	0.80
	jfreechart	0.85	0.75	0.80	0.88	0.83
	Average	0.79	0.82	0.86	0.81	0.81
Multi-Layer Perception Model	*mixprojs*	0.92	0.76	0.79	0.93	0.83
	comcodec	0.70	0.76	0.75	0.69	0.73
	linq4j	0.90	0.78	0.87	0.94	0.90
	comtext	0.71	0.85	0.85	0.71	0.77
	jfreechart	0.75	0.78	0.79	0.77	0.78
	Average	0.79	0.78	0.81	0.80	0.80

only with fixed size test models but it can also perform efficiently on increasing size of test models.

We perform model accuracy and validation loss on 100 epochs as shown in Figure 9 and Figure 10. In Figure 9, the accuracy of train and test models start at 0.75 and 0.80 which gradually increases when epoch size increases. The accuracy of both models becomes stable after 60 epochs where train model accuracy is 0.97 and test model accuracy is 0.95 respectively. Figure 10 shows the validation loss on train and test models where validation loss starts at 0.50 and 0.45 which gradually reduces when epoch size increases. The validation loss becomes stable after 80 epochs where

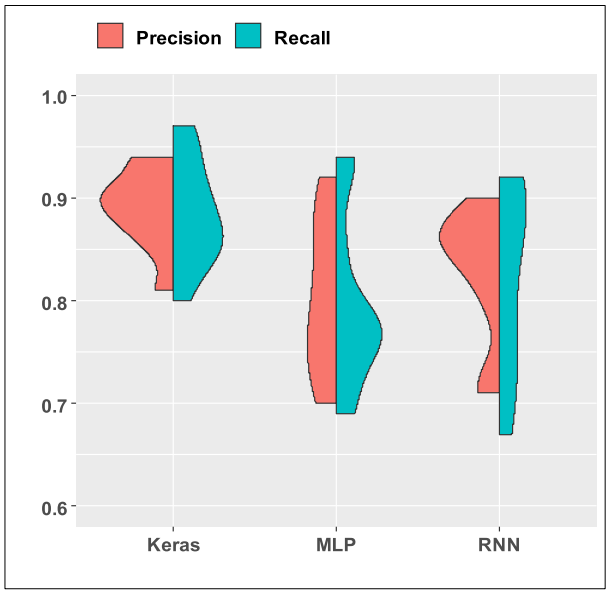

FIGURE 8. Comparison of performance on precision and recall values based on deep learning models.

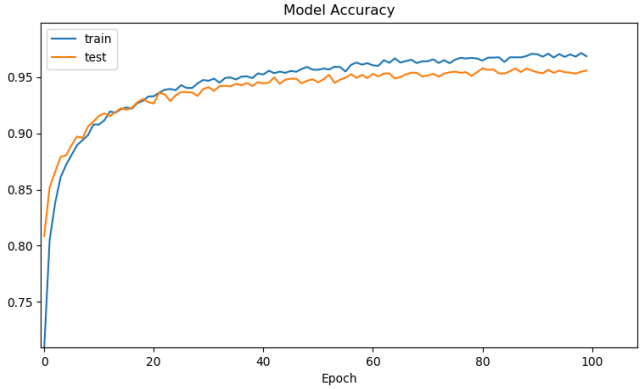

FIGURE 9. The train and the test model accuracy with respect to increasing size of epochs.

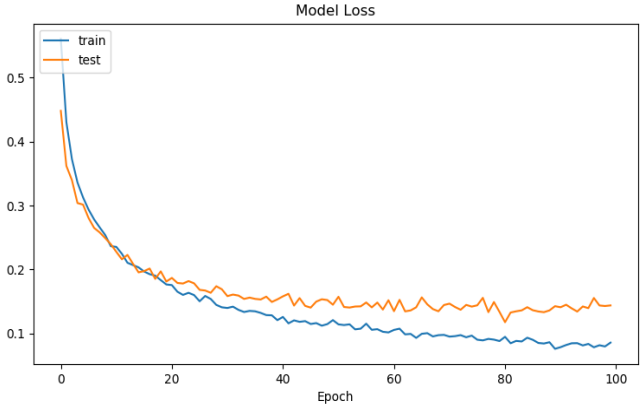

FIGURE 10. The train and the test model loss with respect to increasing size of epochs.

validation loss of the train model is up to 0.10 whereas the validation loss of the test model is up to 0.20 respectively. The analysis on epoch sizes shows that the proposed deep learning model is efficiently fine-tuned on dense and dropout layers along with activation methods and size of neurons.

3) EVALUATION ON TWO TAILED Z-TEST

In the final experiment, we measure the significance of each principal component from PC1 to PC9 using the two tailed Z-test. The two tailed Z-test is a statistical hypothesis to estimate the region of rejection on both sides of the sample distribution [45]. For instance, during the prediction of mutants into *alive* and *killed* classes, the probability of rejection lies on both sides. A *killed* mutant can be incorrectly classified as *alive,* and an *alive* mutant can also be incorrectly classified as *killed* other than the correct classification of *killed* and *alive* mutants. The two tailed Z-test assessment uses the mean and the variance of sample distribution by estimating the deep learning model to calculate the significance of each PC. The probabilities of two tailed Z-test on five programs along with the combined train set is shown in Table 9. The confidence level of each PC is measured by subtracting the probability value from 1. For instance, the highest probability is achieved by the *comcodec* program which is 0.9259859 in PC4. It means the confidence level of PC4 for the *comcodec* program is 7.41% only. The highest confidence level is achieved by the *mixprojs* which is closer to 0 for all PCs. In the *comcodec* program, PC2, PC3, PC5, PC6, PC8 and PC9 have highest confidence level. In the *linq4j*, PC1, PC2 and PC8 achieve the highest confidence level. The *comtext*, *jfreechart* and *combined* programs achieved high confidence level in 3, 7 and 6 PCs respectively. From the observations of two tailed Z-test, we can conclude that all the selected PCs have high confidence in sampling distribution and they have effectively contributed in the prediction of mutant into their respective classes.

V. THREAT TO THE VALIDITY

The threat to internal and external validity may suffer the validly of experimental study, findings and claims.

During program dependence analysis, we found two limitations that may affect the construction of PDGs. Firstly, in opensource programs, unknown method calls are commonly used. The unknown method call further deepens the complexity of PDG which may require more computation cost for feature extraction. To overcome this threat, we used parallel processing on multiple mutants using pool instance a multi-threading feature provided in Python. Secondly, complex data structures are widely used in opensource projects. If a data structure has more than one member variable assignments, then it may become difficult for PDG to determine which variables are directly affected by mutations. To overcome this threat, we use the Eclipse JDT toolkit to implement the feature extraction model. Eclipse JDT can perfectly handle the complexity of Java source codes which may reduce this threat.

Many researchers use different types of mutant generation tools and mutant operators in mutation testing studies. In this paper, we selected Mujava to generate mutants. The selection

TABLE 9. The predictor variables (PC probabilities) are generated using Two Tailed Z-Test. The lower value indicates the higher confidence level and greater significance of principal components for prediction of mutant classes (killed or alive).

PC\Prog.	*mixprojs*	*comcodec*	*linq4j*	*comtext*	*jfreechart*	*combined*
(Intercept)	*0.0000000000*	*0.000000e+00*	*1.643130e-14*	*0.000000e+00*	*1.365505e-02*	*0.000000e+00*
PC1	*0.0004835669*	*3.252702e-01*	*0.000000e+00*	*3.263960e-03*	*0.000000e+00*	*0.000000e+00*
PC2	*0.0000000000*	*3.841696e-05*	*0.000000e+00*	*0.000000e+00*	*0.000000e+00*	*0.000000e+00*
PC3	*0.0000000000*	*0.000000e+00*	*2.169528e-01*	*0.000000e+00*	*5.144990e-05*	*0.000000e+00*
PC4	*0.0000000000*	*9.259859e-01*	*3.014757e-01*	*1.613339e-04*	*0.000000e+00*	*0.000000e+00*
PC5	*0.0000000000*	*0.000000e+00*	*5.087951e-01*	*1.776357e-15*	*0.000000e+00*	*0.000000e+00*
PC6	*0.0000000000*	*0.000000e+00*	*7.674962e-02*	*9.917422e-11*	*0.000000e+00*	*1.536905e-03*
PC7	*0.0000000000*	*3.542635e-06*	*1.084427e-01*	*0.000000e+00*	*0.000000e+00*	*4.138420e-09*
PC8	*0.0000000000*	*0.000000e+00*	*0.000000e+00*	*3.009080e-03*	*5.691170e-07*	*2.220446e-16*
PC9	*0.0000000000*	*0.000000e+00*	*9.120736e-02*	*4.177401e-01*	*0.000000e+00*	*0.000000e+00*

of mutant operators can constitute another possible threat as a reliable mutation testing system requires good understanding of mutation operators. However, the mutant operators used in this study are widely studied in related work such as "*Sufficient Mutant Operators*".

The test suites used in this study may also cause threat to the validly of prediction results. Different test suites produce different results whereas the quality of test suites can affect the classification and model accuracy. To reduce this threat, we used original test suites written by developers of subject programs. Many test suites have imbalance portion of *killed* and *alive* mutants. However, experimental study shows that the proposed approaches can efficiently handle this threat by producing good classification results even for highly imbalanced programs.

VI. CONCLUSION

Machine intelligence has provided solutions in many domains to solve the cost related problems. In mutation testing, the execution of mutants on test suites is also one of those problems which makes its use rare in the software industry. In this paper, we propose and extensively evaluate mutation testing using the prediction of deep learning models on five Java programs selected from the GitHub repository. Firstly, the feature extraction approach is proposed to exact the features from the mutant programs. Secondly, a deep learning model is designed using the implementation of the TensorFlow framework with Keras API. PCA reduction is applied and principal components are selected to remove noisy data to reduce the high dimensionality. The main idea of this paper is to make mutation testing scalable for big programs without any need for test suite execution or loss of test effectiveness. Therefore, predictions are performed on independent test sets. The comparison of the proposed deep learning model with other deep learning classifiers such as RNN and MLP also showed good effectiveness of scalable mutation testing in the prediction of mutants.

In extended experiments, we evaluated the accuracy and validation loss of both train and test models on 100 epochs. This evaluation also showed that the proposed approach not only works efficiently on fixed size test models but also scalable to the increasing size of mutants. Lastly, a two tailed Z-test is performed on selected principal components to measure the confidence level of each component in the classification of mutants for individual mutant program.

In the future, we will extend our techniques to solve the problems in other domains of mutation testing such as mutant selection and dominator mutant factor using machine intelligence.

ACKNOWLEDGMENT

M. R. Naeem would like to thank his colleague Mr. H. Liu for useful discussion and technical assistance for this paper.

REFERENCES

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2010.

[2] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. ICSE*, Jun. 2014, pp. 435–445.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[4] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *Proc. ISSRE*, Nov. 2015, pp. 46–57.

[5] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proc. ICST*, Mar./Apr. 2014, pp. 153–162.

[6] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, "Traceability for mutation analysis in model transformation," in *Models in Software Engineering* (Lecture Notes in Computer Science), vol. 6626. 2010, pp. 259–273.

[7] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proc. ICSE*, May 2010, pp. 435–444.

[8] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Trans. Softw. Eng.*, vol. 20, no. 5, pp. 337–344, May 1994.

[9] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Proc. ICSTW*, Apr. 2010, pp. 90–99.

[10] J. Zhang, M. Zhu, D. Hao, and L. Zhang, "An empirical study on the scalability of selective mutation testing," in *Proc. ISSRE*, Nov. 2014, pp. 277–287.
[11] L. Madeyski, "The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment," *Inf. Softw. Technol.*, vol. 52, no. 2, pp. 169–184, Feb. 2010.
[12] P. K. Singh, O. P. Sangwan, and A. Sharma, "A systematic review on fault based mutation testing techniques and tools for Aspect-J programs," in *Proc. IACC*, Feb. 2013, pp. 1455–1461.
[13] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, Dec. 1995.
[14] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient m utant operators a. Jefferson Offutt Ammei lee George mason University," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, Apr. 1996.
[15] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Softw., Test., Verification Rel.*, vol. 11, no. 2, pp. 113–136, Jun. 2001.
[16] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proc. ICSE*, May 2008, pp. 351–360.
[17] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proc. ISSTA*, Jul. 2013, pp. 224–234.
[18] W. E. Wong, A. P. Mathur, and J. C. Maldonado, "Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness," in *Proc. Softw. Qual. Productiv.*, 1995, pp. 258–265.
[19] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 4, pp. 371–379, Jul. 1982.
[20] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Proc. STVA*, Jul. 1988, pp. 152–158.
[21] M. Singh and V. M. Srivastava, "Extended firm mutation testing: A cost reduction technique for mutation testing," in *Proc. ICIIP*, Dec. 2017, pp. 604–609.
[22] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. ISSTA*, Jun. 1993, pp. 139–148.
[23] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a mimd computer," in *Proc. ICPP*, Aug. 1992, pp. 257–266.
[24] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. ISSTA*, Jul. 2013, pp. 235–245.
[25] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *Proc. ISSTA*, Jul. 2012, pp. 331–341.
[26] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2018.
[27] F. Ullah, H. Naeem, and S. Jabbar, "Cyber security threats detection in Internet of Things using deep learning approach," *IEEE Access*, vol. 7, pp. 124379–124389, 2019.
[28] S. Zafar, S. Jangsher, O. Bouachir, M. Aloqaily, and J. B. Othman, "QoS enhancement with deep learning-based interference prediction in mobile IoT," *Comput. Commun.*, vol. 148, no. 15, pp. 86–97, Dec. 2019.
[29] K. Z. Haider, K. R, Malik, S. Khalid, T. Nawaz, and S. Jabbar, "Deepgender: Real-time gender classification using deep learning for smartphones," *J. Real Time Image Process.*, vol. 16, no. 1, pp. 15–29, Feb. 2019.
[30] M. Aloqaily, I. A. Ridhawi, H. B. Salameh, and Y. Jararweh, "Data and service management in densely crowded environments: Challenges, opportunities, and recent developments," *IEEE Commun. Mag.*, vol. 57, no. 7, pp. 81–87, Apr. 2019.
[31] S. Otoum, B. Kantarci, and H. T. Mouftah, "Empowering reinforcement learning on big sensed data for intrusion detection," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–7.
[32] S. Otoum, B. Kantarci, and H. T. Mouftah, "On the feasibility of deep learning in sensor network intrusion detection," *IEEE Netw. Lett.*, vol. 1, no. 2, pp. 68–71, Jun. 2019.
[33] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science), vol. 4963, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, pp. 337–340. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24
[34] F. Perin, L. Renggli, and J. Ressia, "Ranking software artifacts," in *Proc. FAMOOSR*, vol. 120, 2010, pp. 1–4.
[35] T. H. Haveliwala, "Topic-sensitive PageRank: A context-sensitive ranking algorithm for Web search," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 784–796, Jul./Aug. 2003.
[36] K. Jalbert, "Predicting mutation score using source code and test suite metrics," in *Proc. 1st Int. Workshop Realizing AI Synergies Softw. Eng. (RAISE)*, Jun. 2012, pp. 42–46.
[37] I. T. Jolliffe and J. Cadima, "Principal component analysis: A review and recent developments," *Philos. Trans. Roy. Soc. A*, vol. 374, Apr. 2016, Art. no. 20150202.
[38] K. K. Bharti and P. K. Singh, "Hybrid dimension reduction by integrating feature selection with feature extraction method for text clustering," *Expert Syst. Appl.*, vol. 42, no. 6, pp. 3105–3114, Apr. 2015.
[39] J. Hernandez, J. A. C-. Ochoa, and J. F. M-. Trinidad, "An empirical study of oversampling and undersampling for instance selection methods on imbalance datasets," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (Lecture Notes in Computer Science) vol. 8258. 2013, pp. 262–269.
[40] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. OSDI*, 2016, pp. 265–283.
[41] A. Gulli and S. Pal, *Deep Learning With Keras*. Birmingham, U.K.: Packt, 2017.
[42] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[43] A. Tato and R. Nkambou, "Improving ADAM optimizer," in *Proc. ICLR*, 2018, pp. 1–4.
[44] G. Fraser, M. Staats, and P. McMinn, "Does automated white-box test generation really help software testers?" in *Proc. ISSTA*, Jul. 2013, pp. 291–301.
[45] M. Adam and S. J. Miller, "Tests of hypotheses using statistics," in *Mathematics Department*, vol. 2912. Providence, RI, USA: Brown Univ., 2006, pp. 1–32.

MUHAMMAD RASHID NAEEM received the bachelor's degree from International Islamic University, Pakistan, in 2012, and the master's degree from Chongqing University, China, in 2015, all in software engineering. He is currently pursuing the Ph.D. degree with Sichuan University, China. He is also the author of various research articles published in reputed journals and conferences from Elsevier, Springer, Wiley, and the IEEE publishers. His current research interests include software mutation testing, static analysis, and software testing using machine intelligence.

TAO LIN received the master's degree, in 2003, and the Ph.D. degree from Japan, in 2007. He was a Postdoctoral Researcher and a Guest Lecturer with Waseda University, Japan. He joined the School of Computer Science, Sichuan University, as a Talented Person and established human–computer interaction and the Digital Media Laboratory. So far, he has published more than 30 research articles in reputed journals and conferences at home and abroad. He has Hosted and Participated in a Number of the Ministry of Education, Sichuan Science and Technology Support, and the Japan Society for the Promotion of Science programs. He is currently a full-time Professor with Sichuan University, China. His main research interests include software testing, HCI, automatic usability testing, game intelligence, and so on.

HAMAD NAEEM received the B.E. degree in computer systems engineering from Bahauddin Zakariya University, Pakistan, in 2012, the M.E. degree in software engineering from Chongqing University, Chongqing, China, in 2016, and the Ph.D. degree in software engineering from Sichuan University, Sichuan, China, in 2019. He is currently serving as an Associate Professor with the Department of Computer Science, Neijiang Normal University, Neijiang, China. His research work is published in various renowned journals of Elsevier, Springer, Wiley, MDPI, and the IEEE. His research interests include cybersecurity, malware analysis, code clone, and program analysis. He received the Outstanding Master Student Award from Chongqing University, Chongqing, in 2016.

FARHAN ULLAH received the B.S. degree in computer science from the University of Peshawar, Pakistan, in 2008, and the M.S. degree in computer science from CECOS University Peshawar, Pakistan, in 2012. He is currently pursuing the Ph.D. degree in computer science from the School of Computer Science, Sichuan University, Chengdu, China. His research work is published in various renowned journals of Springer, Elsevier, Wiley, MDPI, and Hindawi. His research interests include software similarity, information security, and data science. He received the Research Productivity Award from the COMSATS Institute of Information Technology (CIIT), Sahiwal, Pakistan, in 2016.

SAQIB SAEED received the B.Sc. degree (Hons.) in computer science from the International Islamic University Islamabad, Pakistan, in 2001, the M.Sc. degree in software technology from the Stuttgart University of Applied Sciences, Germany, in 2003, and the Ph.D. degree in information systems from the University of Siegen, Germany, in 2012. He is currently an Assistant Professor with the Department of Computer Information Systems, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia. His research interests include human-centered computing, computer-supported cooperative work, empirical software engineering, and ICT4D. He is a member of the advisory boards of several international journals and a Guest Editor of several special issues. He is a Certified Software Quality Engineer from the American Society of Quality.