IAES International Journal of Artificial Intelligence (IJ-AI)
Vol. 14, No. 2, April 2025, pp. 894~906
ISSN: 2252-8938, DOI: 10.11591/ijai.v14.i2.pp894-906

A novel approach to enhancing software quality assurance through early detection and prevention of software faults

Deepti Rai, Jyothi Arcot Prashant
Department of Computer Science and Engineering, Faculty of Engineering and Technology, Ramaiah University of Applied Sciences, Bangalore, India

Article Info

Article history:

Received Nov 1, 2023
Revised Nov 15, 2024
Accepted Nov 24, 2024

Keywords:

Code analysis quantifiers
Inter-project adaptability
Line-level fault prediction
Recurrent neural network
Software defect

ABSTRACT

The current manuscript presents a predictive mechanism towards analyzing software defects by developing a line-level fault prediction technique. Current methodologies rely on customized attributes and overlook the sophisticated structural and semantic characteristics inherent in programming languages. This oversight often led to suboptimal defect identification, as code defects are intricately scrambled with their contextual environment. Moreover, conventional software defect prediction (SDP) strategies, typically focusing on larger code units such as modules or classes, impede precise error localization. To address these challenges, this study proposes an automated scheme utilizing a recurrent neural network (RNN) with an attention layer to analyze line-level quantifiers within the code, such as the number of pairwise operations and single operand operators. The efficacy of this learning-driven scheme is validated through comprehensive experiments conducted on several C++ programs. The experimental results demonstrate a 95.8% recall, 83.12% precision, and 90.35% accuracy in identifying fault-prone lines within a testing dataset. These outcomes confirm the effectiveness of proposed SDP scheme in accurately identifying the defects and highlighting its inter-project capabilities, exhibiting the model's adaptability across different software projects.

Corresponding Author:

Deepti Rai
Department of Computer Science, Faculty of Engineering and Technology
Ramaiah University of Applied Sciences
Bangalore, India
Email: deeraisecond@gmail.com

1. INTRODUCTION

Software is essential to modern life, underpinning the critical infrastructure of communications networks, financial systems, transportation, and beyond. In the rapidly evolving scenario of software development, integrity and reliability of software systems have become paramount [1]. The increasing complexity of software, the potential for defects and, as a result, the need for effective software defect prediction (SDP) strategies have become more important than ever. Software defects, or bugs, are unexpected deviations from expected functionality, which can cause anything from minor inconvenience to catastrophic failures, security breaches, and significant financial loss, reputational damage, and even threats to critical systems for human life [2]. A report by the consortium for Information & software quality (CISQ) estimated that poor software quality caused economic losses of $2.84 trillion in the United States in 2018 alone, equivalent to about 1.5% of the country's gross domestic product (GDP) that year [3]. These damages involve a variety of costs, including lost productivity, downtime due to system failures, security breaches, and litigation expenses. This worrying statistic highlights the critical need for effective and early SDP methods.

Early detection and prevention of software defects becomes critical due to several key factors. The primary factor is that the modern software systems are no longer limited to traditional desktop applications [4]. Now they include web-based applications, mobile applications, cloud computing and the internet of things (IoT). Software is ubiquitous in contemporary society and requires a proactive approach to ensuring its quality and reliability. Secondary factor is that the software systems are becoming more and more complex [5]. This complexity increases the risk of potential flaws in the code base, making it difficult to detect them. Tertiary, software failures can have serious financial and reputational consequences. Service disruptions, lost revenue, litigation costs and data breaches can all have a significant impact on businesses and organizations. Last but not least, users' expectations for software performance and reliability have never been so high [6], [7]. Software defects will reduce user trust and satisfaction, and may prompt users to switch to competing platforms [8], [9]. Traditional SDP approaches, while beneficial, have significant limitations. These methods mainly rely on hand-crafted feature engineering methods and focuses on only broader units of analysis such as modules or categories. Such approaches often ignore significant semantic and structural aspects of programming languages, leading to gaps in highlighting the exact location of defects [10]. Furthermore, these methods are often labor-intensive and inefficient, especially in large-scale and complex software systems [11]. As a result, there is an increasing need for more precise and automated SDP technologies that can operate at a more granular level and adapt to various project environments. While there is a clear need for early detection and prevention of software failures, achieving this goal is challenging. The complexity of software systems, the vastness of program code libraries, and the diversity of potential fault types pose huge obstacles [12]. Furthermore, the dynamic nature of modern software development along with agile methods and continuous integration make fault detection more complex [13]. Therefore, the proposed research work introduces a novel computational scheme specifically targeted at line-level fault prediction. The main challenges addressed by the proposed scheme are the lack of accuracy of existing SDP methods and the need for automated systems that can adapt to different software projects without requiring extensive manual intervention. This approach is based on the consideration that more granular analysis at the line-of-code level can significantly improve the accuracy and efficiency of defect prediction.

The recent state-of-the-art methods have increasingly leveraged machine learning techniques due to their ability in processing large datasets and identifying complex patterns. This trend is evident in the work of researchers in [14], the researchers employed fuzzy-Adaboost and LogitBoost algorithms in an empirical study to predict software defects. These boosting algorithms enhance the predictive power by combining weak learners to form a more robust model. Qasem *et al.* [15] compared the efficacy of two deep learning models, multi-layer perceptron (MLP) and convolutional neural network (CNN), in exploring key factors influencing predictive modeling in SDP tasks. These approaches show evolving nature of SDP schemes, and highlights the importance of efficient learning model for precise SDP tasks. However, the performance of SDP is not solely dependent on the choice of machine learning or DL models; it is also significantly influenced by the quality of the training dataset, which often suffers from class imbalance. Therefore, recognizing the limitations posed by class imbalance in training datasets, Bejjanki *et al.* [16] introduced a class imbalance reduction algorithm generating synthetic data points in the minority class to balance defect and non-defect data samples, addressing the prevalent issue of class imbalance in software datasets. This approach of class balance is further refined by Feng *et al.* [17] who proposed a complexity-aware oversampling algorithm. Their approach, which pairs defective instances based on complexity, suggested a more nuanced method of oversampling, adding depth to the solutions for class imbalance. Complementing these oversampling strategies, Goyal [18] approached the issue from a different angle with an under-sampling method using the nearest neighbor algorithm. This method provided an alternative solution, suggesting the need for diverse approaches in addressing dataset imbalances. Apart from handling class imbalance, few researchers highlighted the importance of preprocessing and feature extraction techniques. The research work in this direction is carried out by Turabieh *et al.* [19], where the authors employed various optimization techniques such as genetic algorithm (GA), particle swarm optimization (PSO), and ant colony optimization (ACO) for feature ranking and extracting optimal feature subsets. However, these optimization model needs adjusting or configuring suitable parameters and provide solution at the cost of higher computation resources. Similarly, the work of Alsghaier and Akour [20] applied PSO and GA to the NASA dataset for feature selection, training a support vector machine (SVM) to predict faults. Further expanding on feature selection, Tumar *et al.* [21] implemented moth flame optimization (MFO) followed by adaptive synthetic sampling (ADASYN) for up-sampling. An application of golden jackal optimization (GJO) is introduced in the work of [22]. Following feature selection, various supervised classifiers were trained to predict software defects. An interesting empirical work by Balogun *et al.* [23] examined various feature selection techniques to analyze their impact on software fault predictive task performance. Recent works also reported the problem of capturing programs' syntax and semantic information. To handle this problem, semantic learning that combines word embedding and an RNN model is suggested by [24], and an attention-based RNN model is presented in [25]. Mafarja *et al.* [26] used random forest (RF) classifier, enhanced with binary whale optimization algorithm for an optimal feature selection. Li *et al.* [27] introduces a framework utilizing deep learning for detecting vulnerabilities in C/C++ programs by obtaining program representations accommodating syntax and semantic information. Feng *et al.* [17] address class imbalance issues by generating synthetic instances, improving diversity without compromising model effectiveness. A different SDP method is presented by [28] that maximizes the proportion of found bugs during code inspection using a linear regression model and a re-ranking strategy, significantly finding more bugs with fewer initial false alarms, although this approach has its limitations in terms of scalability and higher computing cost. Hence, based on the review of literature it can be seen that SDP remains a challenging field despite significant advancements in software engineering practices due to following issues:

– Imbalanced data: in real-world datasets, defect-prone code often represents a minority compared to defect-free code. This data imbalance can pose challenges for machine learning models, leading to biases towards the majority class (defect-free) and potentially missing subtle defect patterns.
– Data noise and inconsistencies: software code often contains inconsistencies, formatting variations, and incomplete information. This noise can negatively impact the performance of SDP models requiring careful data cleaning and preprocessing.
– High dimensionality: the software code is represented using a large number of features, leading to the curse of dimensionality and potentially hindering model training and interpretation. Feature selection and dimensionality reduction techniques are required to address this issue.

Existing research works on SDP have made significant progress to address above mentioned SDP design challenges especially using machine learning and meta-heuristic optimization methods. Despite significant advancements in SDP techniques, several crucial research gaps remain highlighted as follows:

– Repetitive nature of methodology: It has been analyzed that most of current researches in literature are mostly similar and depended on established methodologies, lacks of novelty in their design.
– High computational cost: many studies in SDP employ sophisticated optimization methods to achieve optimal feature selection. While optimization is vital for model effectiveness, it often leads to a significant increase in computational resources. This raises the need for models that not only perform well but also maintain computational efficiency.
– Implementation design specificity: many current SDP approaches are designed with a narrow focus, designed to specific problem or datasets. Such specificity restricts the models' scalability and applicability across different domains and codebases. The development of versatile models, adaptable to various contexts, is thus a critical area for future research.
– Lack of optimization: the current SDP models often lack robustness in their design and optimization processes. This lack of robustness renders them less capable of effectively handling the dynamic and complex nature of software codes, subsequently impacting their effectiveness and reliability.

In order to address the above-mentioned gaps, this paper introduces a novel computational approach combining advanced learning algorithms and efficient data processing. The methodology emphasizes line-level analysis using natural language processing (NLP) and recurrent neural networks (RNN) with attention layer. Initially, it involves critical data preprocessing to optimize the training dataset. Subsequently, a comprehensive set of line-level quantifiers is extracted to assess the code's syntactic and semantic features. This in-depth analysis, unique in SDP, significantly enhances precision and reliability. The method's automated nature ensures adaptability across diverse software projects and languages with minimal manual intervention. Its line-level approach provides improved defect localization within code modules, offering more precise fault identification and quicker correction. This granular analysis facilitates targeted defect detection and resolution, ultimately reducing computational time and resources needed for defect management. The key highlights of this paper are:

– This research introduces a novel SDP methodology focusing on line-level analysis and implements essential preprocessing of input code dataset, optimizing it for more accurate and enhanced the precision in defect localization.
– Develops and utilizes a detailed set of line-level quantifiers, assessing both syntactic and semantic aspects of code, thereby significantly improving the reliability of defect detection.
– Implements Bidirectional long short-term memory (Bi-LSTM) a special class of RNN with attention layer and customized loss function to make training process more robust, thereby reduced misclassification rates and training cost.
– Features an automated approach that adapts easily across various software projects and programming languages, reducing the need for manual intervention, and increasing its applicability.

2. METHOD

This section of the research paper details the design and methodology of a the proposed system designed to predict software defects within C++ code. The research methodology adopted for proposed SDP system is designed analytically, composed for different computing blocks which are highlighly synchronized and integrated in sequential manner emphasizing line-level defect prediction. The propsoed SDP system employs advanced machine learning technologies, harnessing the capabilities of both exisitng preprocessing algorithms and deep learning networks to learn patterns indicative of defects and analyze code lines at various levels of granularity. This precision significantly enhances the efficiency of software testing by directing testing efforts to the specific lines of code that are most likely to contain defects, rather than broader code segments. The implementation of line-level SDP is realized through a sophisticated application of NLP and RNNs, which are effective at capturing and interpreting the sequential nature of code fault prediction. By treating each line of code as a distinct unit of analysis, the system is capable of identifying critical and context-specific anomalies that exisitng approaches often overlooked. The system is designed to serve a multiple purpose viz: i) to enhance the efficiency of software testing by preemptively identifying fault-prone code lines and ii) to aid developers in mitigating potential vulnerabilities before deployment. The schematic architecture and methodology flow of the propsoed SDP system is shown Figure 1.

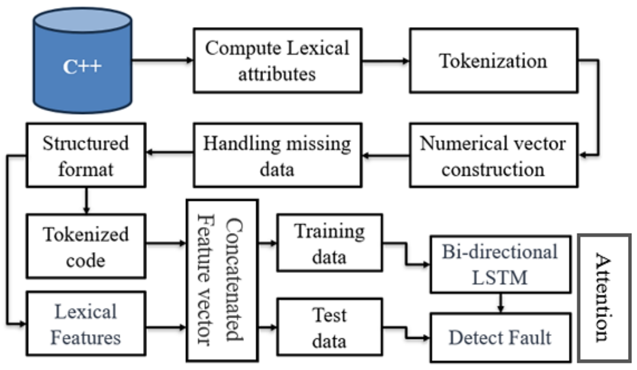


Figure 1. Illustrating block-based architecture of the propsoed methodology for SDP

As shown in Figure 1 the proposed system considers its input from the database consists of several codes configured in C++ programming language. Once the all the input codes are loaded in computing environment, the proposed system computes all the possible lexical attributes. The proposed system incorporates a tokenization phase that transforms code into a sequence of discrete, analyzable elements, followed by the extraction of lexical features that are critical for line-level analysis. These features serve as the input to the RNN, which is trained to recognize patterns associated with software defects at the line level. The next module of the proposed system executes data transformation process, where tokenized code is turned into numerical vectors suitable for training RNN model. The sub-sequent procedure of the proposed system applies data preprocessing operation stage where any missing or inconsistent data within the code files is handled to ensure data quality and readiness for further analysis. This part of the system also focuses on making the data uniform in shape by padding or truncating each line of code within a file to achieve uniformity across all files.

The obtained tokenized data and lexical attributes of the code is then concatenated in a single vector serving as final dataset. This vector serves as the final dataset, ready for the next crucial phase of data compilation and normalization. During this phase, the dataset is divided into training and testing sets, preparing it for the model training process. The proposed study them implements Bi-LSTM model which is a special type of RNN to process sequential data such as code, which is key for detecting complex patterns associated with software defects. Furthermore, the use of Bi-LSTMs allows the system to capture context in both forward and backward directions within the code, offering a more effective and fine-grained analysis than traditional, models. The study introduces a customized loss function that assigns different weights to classes, thereby effectively mitigating the bias towards the majority class (commonly the non-defective class) and enhancing the model's ability to detect the rarer defective lines of code. The final step involves validating the system using the testing data set. Here, the trained model is employed to predict defects in new, unseen code submissions.

2.1. Data loading and structuring

Since the database consists of many code files and to read and preprocess data from the database store, it is necessary to handles potential irregularities, and performs initial preprocessing for transforming raw data into a structured format that can be further processed and analyzed. The process of reading data from the database store can be described through a series of mathematical operations that transform the raw C++ code files into a structured format for analysis. The entire process of data loading in computing environment as structure data-frame involves reading all code files from datastore $\mathcal{A} = \{f_1, f_2, f_3, \cdots f_n\}$, where each f_i represents a code file contained within the datastore $\mathcal{A}$. The extraction and preprocessing function $\mathcal{P}$ operate on the datastore $\mathcal{A}$ and applies a series of transformations to each file f_i to produce a preprocessed dataset D such that: $D = \mathcal{P}(\mathcal{A}) = \{\mathcal{P}(f_1), \mathcal{P}(f_2), \mathcal{P}(f_3) \cdots, \mathcal{P}(f_n)\}$. Each individual preprocessing function $\mathcal{P}(f_i)$ can be broken down into three main stages: i) extraction $\mathcal{E}(f_i)$ extracts the file f_i from the archive, ii) normalization $\mathcal{N}(f_i)$ handles irregularities such as inconsistent line breaks, character encoding issues, or varied indentation styles and normalizes f_i to a consistent format, and iii) tokenization preparation $\mathcal{F}(f_i)$ prepares f_i for tokenization by segmenting the code into logical units that can be tokenized, like lines or statements.

The (1) describes the preprocessing function $\mathcal{P}$ for each file. f_i. After processing, dataset D is a collection of preprocessed code files ready for tokenization and feature extraction. Each $\mathcal{P}f_i$ in D is a structured representation of the original file, transformed into a format that is amenable to analysis by machine learning algorithms.

$$\mathcal{P}(f_i) = \mathcal{T}(\mathcal{N}(\mathcal{E}(f_i)) \qquad (1)$$

Algorithm 1. Data loading and structuring
```
Input: $\mathcal{A} = \{f_1, f_2, f_3, \cdots f_n\}$
Output: Processed data set $DF$
Start
    1.  Initialize an empty list $D = []$ to store processed data.
    2.  For each file $f_i \in \mathcal{A}$:
    3.      read $f_i$ as a CSV file and append to $D$
    4.  For each data frame $df_i \in D$
    5.      if $df_i$ is empty, go to the next data frame
    6.        Let $A = df_i$ values be the matrix representation of $df_i$
    7.      Copy $A$ to $B$ (i.e., $B = A$)
    8.  Modify $B$ to include both correct and defect class
    9.        $B = conc(B_{:,:-1}, flip(B_{:,-1:}), B_{:,-1:})$
    10. For each row j in B:
    11.       If $\sum\left(A_{j-range_{n:j}+range_{n,}-1}\right) > 0$:
    12.          Set $B_{j,-1} = 1$ and $B_{j,-2} = 0$.
    13. Convert non-code elements from string to numeric where possible:
    14.     For each element $B_{x,y}$:
    15.        If $B_{x,y}$ is a string, try converting to float.
    16.         Else set $B_{x,y} = -x$ //(marker to indicate non-convertible data and can be any
        random vale)
    17. Convert B back to a data frame with appropriate column names
    18. Replace '#empty' with NaN and drop rows with NaN values.
    19. Append the cleaned data frame to the final result DF
    20. Return the final processed data set DF
End
```

The Algorithm 1 shows computing steps to read data and load in a structured data frame approach in a computing environment for subsequent operations tokenization, cleaning, and training in SDP. The algorithm takes an input set $\mathcal{A}$ containing files f_i, and after execution it returns a processed dataset D. The process begins by initializing an empty list D to accumulate the processed data (step-1). It then iteratively reads each file f_i from the set $\mathcal{A}$, treating them as CSV files, and appends the contents to D (step 2-3). The next operation focuses to each individual data frame df_i within D. Here, the algorithm neglects any empty frames, ensuring that only content-rich data is processed further. For every non-empty data frame, the algorithm extracts its matrix representation A, and then duplicates this matrix into a new matrix B, placing the basis for subsequent data transformations (step 4-7). In the next step, the algorithm executes a data transformation operation over matrix B to incorporate both the correct and defective classes. This is achieved through a concatenate operation ($conc$) which appends a flipped or transformed version of the last column alongside the original data. This operation essentially enriches the dataset with both positive and negative aspects of the defect classification, which is critical for a balanced analysis (step 8-9). The proposed algorithm then scrutinizes each row j in matrix B. Here if the sum of a specified range around the defect column exceeds zero, it alters the respective entries in B to ensure that both defect presence and absence are accurately represented (step 10-12). In handling non-code elements within matrix B, the algorithm attempts to convert string entries to floats. Where this conversion is infeasible, it assigns a marker value (-x, for instance -10) to signify non-convertible data. This step is crucial in standardizing the data type across the dataset, thereby facilitating smoother computational analysis (step 13-16). In the sub-sequent steps, the matrix B is returned to a data frame format with comprehensive and accurately code features (step 17). The algorithm also cleanses the data by replacing '#empty' markers with not a number (NaN) and subsequently discarding any rows overloaded with NaN values (step-18). This cleaning operation ensures data integrity and relevance. Finally, the cleaned data frame is appended to the DF, which represents the fully processed dataset (step 19-20). It is to be noted that here DF is a list of data frames i.e., list of structured code having row and columns (lexical features)

2.2. Tokenization

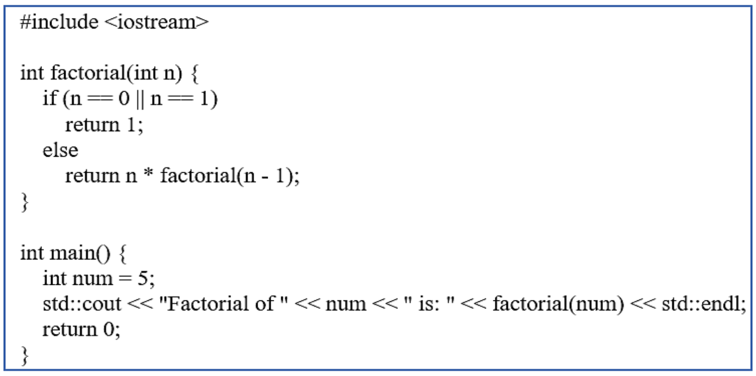
Tokens are the basic building blocks used in NLP and machine learning tasks. Tokenization is the process of breaking down a text or a sequence of characters into smaller units, called tokens. In the context of programming languages or code, tokenization refers to breaking down the source code into format that can be analyzed quantitatively like individual tokens, where each token represents a meaningful element in the code. The Algorithm 2 presents an implementation of computing steps for converting a collection of structured code files into a tokenized format, which is essential for DF. The algorithm considers an input data frame DF, which comprises a list of several data frames representing structured code with rows and columns. The output of this algorithm is a tokenized dataset $\mathcal{T}$ along with the average number of lines per file μ and the maximum number of lines in any single file max_n.

Algorithm 2. Tokenization

```
Input: $DF$ comprising list of several data frame a structured code having rows and columns
Output: Tokenized data $\mathcal{T}$, average number of lines $\mu$ and maximum number of lines $max_n$
Start
    1.  Initialize an empty vector $\mathcal{T} = []$ for storing tokenized data.
    2.  Initialize $\mu$ and $max_n$ and set both to zero
    3.  For each code file $C_i \in DF$:
    4.       if $C_i$ is empty (i.e., $|C_i| = 0$), skip to the next file
    5.       Initialize a temporary vector temp = [ ]
    6.       Update $\mu$ and $max_n$ based on the number of lines in $C_i$
    7.        For each line $L_j$ in $C_i$:
    8.             Get token: $t = \mathfrak{f}_1(L_j)$ // where $\mathfrak{f}_1(\cdot)$ is function for tokenization
    9.              Append $t$ and and corresponding class information y to vector temp
    10.       Append vector temp to $\mathcal{T}$
    11. Compute mean: $\mu = \frac{\sum_{i=1}^{n}|C_i|}{DF}$
    12. Compute $max_n = max_{i=1}^{n}|C_i|$
    13. Return $\mathcal{T}$, $\mu$, $max_n$.
End
```

The first step of the algorithm initializes an empty vector $\mathcal{T}$ that will eventually store the tokenized data (step 1). In the sub-sequent steps, two numerical variables μ and max_n are initialized and set to zero. These variables will be used to calculate the average and maximum number of lines in the code files, respectively (step 2). Next, the algorithm iterates over each code file C_i within the data frame DF (step 3). For each file, it first checks if the file is empty $|C_i|$=0. If a file is found to be empty, the algorithm skips it and moves to the next one (step 4). For non-empty files, a temporary vector temp is further initialized to hold the tokenized data for that particular file (step 5). As the algorithm processes each line L_j in the code file C_i it employs a tokenization function $\mathfrak{f}_1(\cdot)$ to tokenize each line of code L_j (step 6-8). Here, the function $\mathfrak{f}_1(\,)$ basically, a kind of lexical analyzer that reads the code and segments it into tokens or block. This function also uses a set of pre-defined patterns (^[a-z, A-Z_][a-zA-Z0-9_]*$, ^if$, ^else$, ^for$, ^\d+\.\d+([eE][-+]?\d+)?$, ^\"(\\.|[^\\"])*\"$ and many more) and rule subjected to keywords, operators, literals, and identifiers that outlines how different segments of the code should be interpreted and tokenized. For instance, when this function encounters a sequence of characters like if, it uses the predefined rules to recognize this as a keyword token. Basically, it uses a regular expression (regex) customized to the C++ language syntax are designed to match various lexical elements of the language, such as keywords, operators, identifiers, and different types of literals.

The function $\mathfrak{f}_1(\cdot)$ returns the tokenized version t of the line L_j. Each token t along with its corresponding class information t (indicating whether the line of code is correct or defect), is then appended to the vector temp (step 6). After processing all lines in a code file, the vector temp, which now contains the tokenized representation of the entire file, is appended to the main vector $\mathcal{T}$. This process is repeated for each file in DF gradually building up the tokenized dataset $\mathcal{T}$ (step 9-10). To better understand the tokenization operation Figure 2 presents a smaple exmaple of C++ code and the potential output from this Algorithm 2.



```
#include <iostream>

int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num = 5;
    std::cout << "Factorial of " << num << " is: " << factorial(num) << std::endl;
    return 0;
}
```

Figure 2. Sample illustartion of C++ code

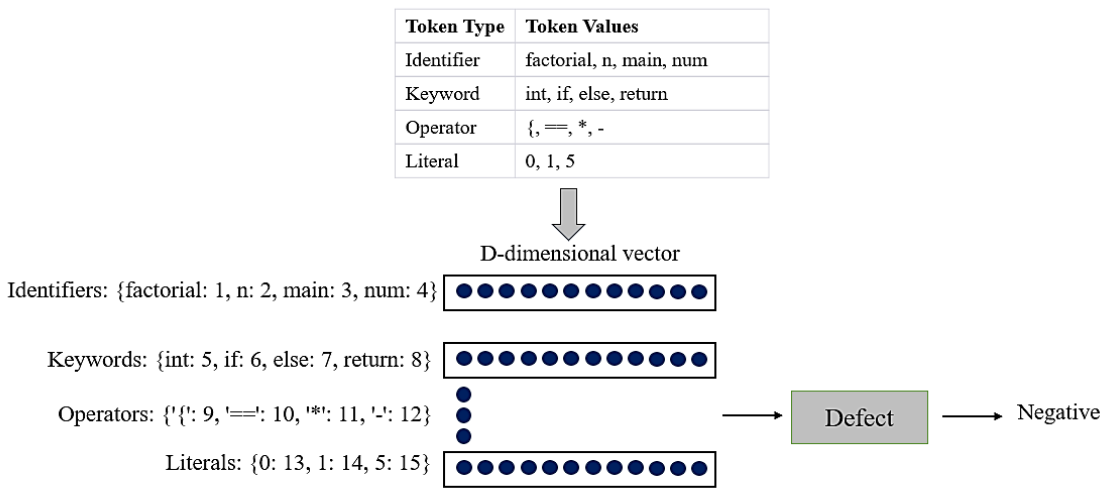
It can be seen that the tokenization process for C++ code in Figure 2 using Algorithm 2 breaks down each line into identifiable elements like keywords, identifiers, literals, and operators as shown in Table 1. This structured tokenization is crucial for further analysis, such as understanding the code's functionality, syntax, and potential areas for defects. It exemplifies how raw code can be converted into a format that's more suitable for analytical processes.

Table 1. Potential output from tokenization (Algorithm 2)

Token type	Token values
Identifier	factorial, n, n, n, main, num
Keyword	int, if, else, return
Keyword/Operator	{, ==
Literal	0, 1, 1, 5, 0
Namespace/Operator	std::cout << "Factorial of " << num << " is: " << factorial(num) << std::endl;
Operator	{, ==, *, -
Parentheses	(,), (,), (,)
Preprocessor	#include, <iostream>
Punctuation	;, ;
Keyword/Operator	return

Once all files in DF have been processed, the algorithm calculates the mean number of lines μ across all files. This is done by summing the number of lines in each file and then dividing by the total number of files in DF (step 11). Similarly, the algorithm determines the maximum number of lines DF found in any single file within DF (step 12). Finally, the algorithm concludes by returning the tokenized dataset $\mathcal{T}$ along with the calculated average number of lines μ and the maximum number of lines (step 13). This tokenized dataset $\mathcal{T}$ is now ready for training the model, providing a structured and analytical representation of the original code files. After tokenization, each token is then represented as a string. However, machine learning models, do not work with raw strings. Therefore, the tokeized data is converted into numerical vector by mapping tokens to numbers means assigning a unique integer to each unique token. For example: int→1, return→2, +→3, a many more as shown in Figure 3.

After mapping token to numerical data, the study creates a function that aims to standardize the shape of each coding file in the obtained data frame DF from algorithm. This involves ensuring that each line has a fixed number of words represented by their tokenized format. The function utilizes truncation for excessively long lines and padding for those that are too short. This standardization is crucial for maintaining consistent input shapes with a fixed size. By applying truncation or padding to the tokenized representations, the study ensures uniformity across lines in coding files, enhancing dataset homogeneity, and facilitating processing. Furthermore, the study proposes concatenating lexical features (columns of the data frame obtained from Algorithm 1) with the tokenized data. This process creates a comprehensive dataset that captures both lexical aspects and higher-level characteristics of the input code. The main objective is to form a matrix R by horizontally concatenating T (tokenized data) and the columns of DF (lexical features). This integration is motivated by the recognition that relying only on lexical tokens in the SDP process may not be sufficient to capture the complete context or intrinsic characteristics of the code. Metrics or features derived from code execution, structural attributes, and metadata can provide valuable information. Therefore, the study combines lexical information from Algorithm 1 with processed tokens obtained from Algorithm 2 features to create a more all-inclusive representation of the code.



Token Type	Token Values
Identifier	factorial, n, main, num
Keyword	int, if, else, return
Operator	{, ==, *, -
Literal	0, 1, 5

D-dimensional vector
Identifiers: {factorial: 1, n: 2, main: 3, num: 4}
Keywords: {int: 5, if: 6, else: 7, return: 8}
Operators: {'{': 9, '==': 10, '*': 11, '-': 12}
Literals: {0: 13, 1: 14, 5: 15}
Defect
Negative

Figure 3. Mapping tokens to numbers

2.3. Learning model

A RNN is a deep learning model designed to process sequential data, such as text, speech, or time series data. Unlike traditional neural networks that treat each input independently, RNNs remember information from previous inputs, giving them an advantage in understanding the context of sequential data. To better understand how LSTM works Figure 4 illustrates the architecture of basic and single LSTM neural unit following gating mechanism.

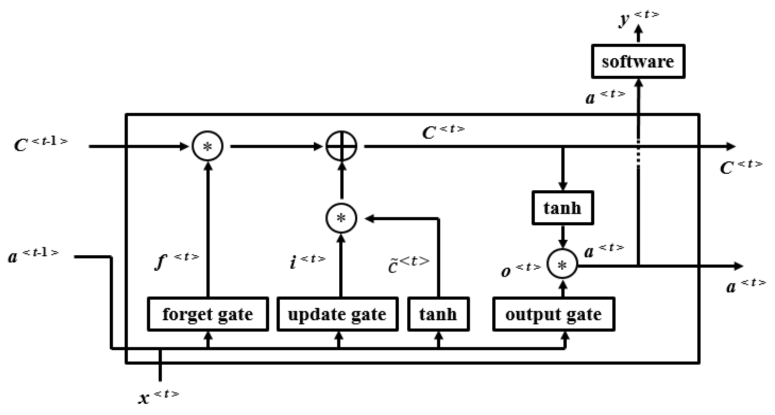


Figure 4. Typical architecture of LSTM cell

The Figure 4 illustrates the typical architecture of an LSTM cell, a specialized unit within RNN designed for processing sequences of data. An LSTM cell is capable of learning long-term dependencies and is particularly useful in tasks that require the understanding of context over time, such as language processing or time series prediction. LSTM networks use a special type of gate mechanism to control the flow of information through the network, which allows them to learn long-range dependencies in the data more effectively than traditional RNNs. At each time step t, the LSTM cell receives two primary inputs: the current input vector $x^{<t>}$ and the output from the previous timestep $a^{<t-1>}$. It also maintains a cell state $\tilde{c}^{<t>}$, which acts as a form of memory. The cell state is carried forward through each timestep, accumulating relevant information throughout the sequence. Within the cell, there are three gates that manage the cell state and the flow of information: the forget gate ($f^{<t>}$), the input or update gate ($i^{<t>}$), and the output gate ($o^{<t>}$).

Each gate applies a sigmoid activation function to weigh its inputs, which range from 0 to 1, effectively controlling the extent to which information is allowed to pass through. The forget gate decides which parts of the previous cell state should be kept or discarded as the sequence progresses. The input gate, in conjunction with the candidate cell state ($\tilde{c}^{<t>}$), decides which new information should be added to the cell state. The candidate cell state is generated by applying a tanh activation function, which helps regulate the information to ensure that the cell state values remain between -1 and 1. These components come together to update the cell state $c^{<t>}$ to its new form. This update is a combination of the old cell state, modulated by the forget gate's output, and the new candidate cell state, scaled by the input gate's activation. This selective update process allows the LSTM to maintain long-range dependencies in the data. Finally, the output gate controls which parts of the cell state will be output as the hidden state $a^{<t>}$ for the current timestep. This output is determined by filtering the cell state through the tanh function to normalize its values, which is then element-wise multiplied by the output gate's activation. The hidden state $a^{<t>}$ captures the LSTM's learned representation at time t and is used both as an output of the current cell and as an input to the next timestep, alongside the updated cell state $c^{<t>}$. The architecture of the LSTM cell, with its gated mechanism, enables it to effectively capture temporal relationships and patterns within sequential data.

The proposed study employed a Bi-LSTM model i.e., LSTM with both forward direction and backward direction. Also, this model is integrated with attention layer that allows the model to dynamically focusing on relevant parts of the input code sequence. The attention mechanism works by computing attention scores that indicate the importance or relevance of each element in the input sequence. These attention scores are then used to compute a weighted sum of the input elements, where elements with higher attention scores contribute more to the output. Therefore, by analyzing line-level quantifiers within the code, such as the number of pairwise operations and single operand operators, the model gains insight into the complex semantic and structural characteristics inherent in programming languages. This learning network is specifically design for the binary classification of sequence data following understanding sequence patterns, dense layers for feature transformation, and regularizations like dropout and batch normalization to ensure robust and efficient training. The proposed study also introduces a custom loss function to minimize the difference between the predicted probabilities and the true labels. The study here presents weighted categorical cross-entropy (WCCE) loss expressed as follows:

$$CCE(Y,P) = -\frac{1}{B}\sum_{b=1}^{B}\sum_{c=1}^{C} Y_{b,c}\log(P_{b,c}) \tag{2}$$

$$WCCE(Y,P) = -\frac{1}{B}\sum_{b=1}^{B}\sum_{c=1}^{C}\sum_{k=1}^{C} Y_{b,c}\log(P_{b,c}) \times W_{c,k} \times M_{b,k} \tag{3}$$

In (2), Y denotes true label matrix of shape [B, C] where B is the batch size and C is the number of classes, P is the predicted probability matrix of the same shape [B, C]. In (3), W denotes the weight matrix of shape [C, C], where Wi,j gives the weight when the true class is i and the predicted class is j.

$$\begin{cases} 1 \; if \; P_{b,c} = max_c P_{b,c} \\ 0 \qquad otherwise \end{cases} \tag{4}$$

The WCCE is an extension of the CCE that allows for this differentiation. By introducing a weight matrix W, the model can be guided to prioritize specific types of errors. Here, $M_{b,k}$ ensures that weights are applied only to the maximum predicted probability for each instance. For Bi-LSTM the function extends by including the time dimension T. The loss is then averaged across all time-steps.

$$WCCE_{LSTM}(Y,P) = -\frac{1}{B\times T}\sum_{b=1}^{B}\sum_{t=1}^{T}\sum_{c=1}^{C}\sum_{k=1}^{C} Y_{b,t,c}\log(P_{b,t,c}) \times W_{c,k} \times M_{b,k,t} \tag{5}$$

Where $WCCE_{LSTM}$ refers to WCCE for LSTM network, process sequences of data, adding a time dimension to the data. Therefore, this incorporates time dimension, thereby ensures that the loss is computed across all time steps, giving a more comprehensive measure of the model's performance over sequences. By implementing this customized approach in loss calculation, the system ensures that the model training is not only focused on accuracy but also on the relevance and significance of the detected defects.

3. RESULTS AND DISCUSSION

The design and dvelopment of the propsoed SDP system is carried out uting python programming langauge and traning of the model is executed in anaconda discrtibution installed on Windows 11 64-bit system. The proposed SDP system leverages a sophisticated Bi-LSTM with attention layer architecture, emperically constructed with multiple layers, where the first layer consists of 180 neurons and includes a dropout rate of 20% to prevent overfitting. The bidirectionality of this layer allows the model to learn dependencies from both past (backward) and future (forward) input sequences, providing a comprehensive understanding of the code context. Afterwards, two additional LSTM layers is implemented with 150 and 100 neurons, respectively. Each incorporates a dropout layer with a rate of 20%, to regularize the model and improve generalization to unseen data. After LSTM layer, the model includes a dense layer with 64 neurons, followed by a additional dense layer with 32 neurons, and 16 neurons. Each of these layers uses the rectified linear unit (ReLU) activation function and batch normalization on its outputs. The ReLU function introduces non-linearity, allowing the model to learn complex patterns, while batch normalization standardizes the outputs to speed up training and improve performance. The final layer of the model is a dense layer with 2 neurons, corresponding to the binary classification task of predicting whether each line of code is defective or not. The training of model is carried for 100 epochs with a mini-batch size of 32, which was emperically decided to acheive balance between computational efficiency in traning process and the ability to reach convergence to an optimal set of weights.

The model is trained on the Code4Bench dataset [29], a multidimensional benchmark repository containing various programming languages. To effectively evaluate the efficiency of the proposed model, this study also considers popular supervised classifiers such as RF, K-nearest neighbor (KNN), and baseline Bi-LSTM models since the proposed contribution is not limited to the introduction of the Bi-LSTM attention model; it also focuses on novel data modelling and feature engineering processes, including precise data structuring, tokenization, and feature mapping. Performance evaluation considers popular classification/prediction metrics such as accuracy, precision, recall, and F1 score. The evaluation has been done for average result of 10-fold cross validation for each metrics. For comparative evaluation with previous work, the proposed study considers similar existing research work by Munir *et al.* [30], where researchers introduced DP-AGL model that integrates of dual RNN models LSTM and gated recurrent unit (GRU) evaluated on the same dataset. Figure 5 demonstrates the performance of different prediction models for identifying defects in software codes.

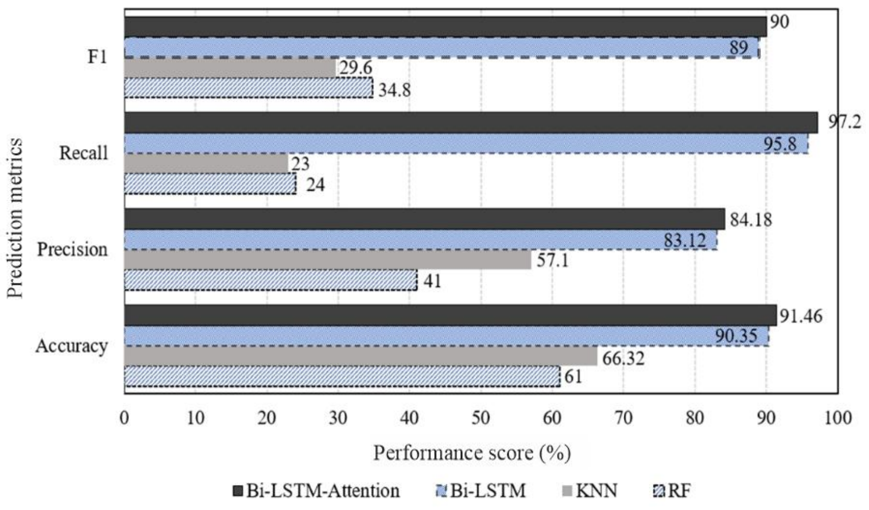


Figure 5. Analysis of prediction models implemented in experimental process

In Figure 5, the analysis is carried out in terms of accuracy, precision, recall, and F1-score. The accuracy measures the proportion of true results among the total number of cases examined, while precision evaluates the proportion of true positive predictions in relation to all positive predictions made by the models. The recall metric measures the ability to identify all actual positives correctly and F1-score on the other hand measures a weighted balanced among precision and recall metric. The graph trend shows that the proposed Bi-LSTM-attention model demonstrates superior performance considering all performance metrics against baseline classifiers. Based on the careful analysis it can be observed that RF classifier achieved an accuracy of 61%, with precision, recall, and F1-score values of 41%, 24%, and 34.8%, respectively. KNN demonstrated improved performance, achieving an accuracy of 66.32%, but with lower precision (57.1%)

and recall (23%) values, resulting in an F1-score of 29.6%. These baseline classifiers getting lowest accuracy, indicating that it may not be as effective at capturing the patterns necessary for defect prediction in this context. The Bi-LSTM model exhibited significantly higher accuracy (90.35%) and balanced precision (83.12%) and recall (95.8%) values, leading to a robust F1-score of 89. The higher performance of Bi-LSTM model indicating a strong capability to correctly identify defect-prone code while minimizing false positives. However, the proposed Bi-LSTM-attention model further retained enhanced performance, achieving an accuracy of 91.46% with precision, recall, and F1-score values of 84.18%, 97.2%, and 90%, respectively. This improvement suggests the effectiveness of incorporating the attention mechanism, which enables the model to focus on relevant parts of the input sequence, thereby enhancing defect prediction accuracy. In order to validate the effectiveness of proposed system, a comparative analysis is conducted with similar existing work [30] as shown in Figure 6.

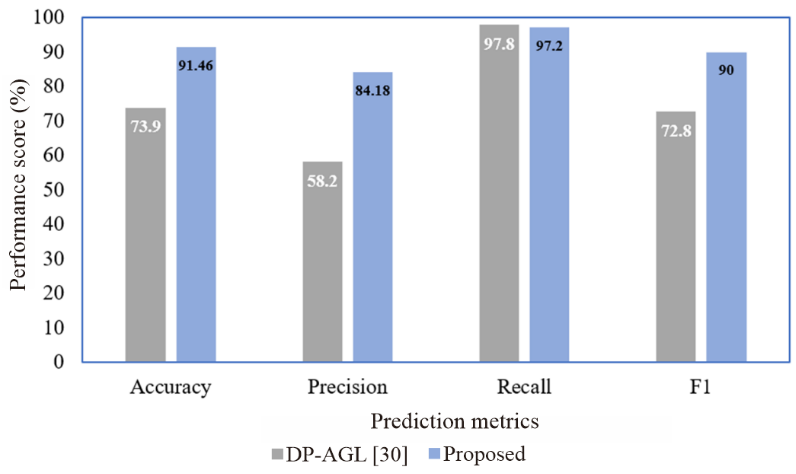


Figure 6. Comparative analysis of prediction models for SDP

The graph trend from Figure 6 demonstrates the superiority of the proposed SDP system over existing model DP-AGL [29] with an accuracy of 91.46%, and F1-score of 90. While DP-AGL demonstrates slightly higher recall (97.8%) due to joint approach of both LSTM and GRU which likely contributes to capturing maximum actual positives class, but fails to capture false positives. The gating mechanisms in GRU and LSTM units help in retaining information over longer sequences, but they may not be as effective as the bidirectional nature of Bi-LSTM in capturing dependencies that span across distant time steps from both past and future states simultaneously. Moreover, the proposed system outperforms DP-AGL in terms of F1-score, achieving a more balanced trade-off between precision and recall, which is critical for practical SDP applications. The overall performance analysis validates the effectiveness of the proposed scheme for SDP due to its sophisticated and comprehensive architecture, capable of learning from both past and future context within a sequence, is particularly suited for the SDP task. Also, the proposed data processing feature extraction process plays an import role in capturing latent and precise attributes that indicates the faults or defects in the software code.

4. CONCLUSION

This research has presented an advance learning scheme in the field of SDP through the implementation of an effective data preprocessing, and feature engineering scheme. The study introduces a unique software code tokenization methodology and effective data processing techniques specifically designed to enhance the model’s predictive capabilities. The proposed tokenization process performs code-live level analysis which separates the software code into quantifiable tokens. By transforming complex code constructs into a machine-interpretable format, the study enables the Bi-LSTM-attention network to effectively learn and predict potential defects with a precise level of accuracy. The precision and recall rates achieved highlights the model's enhanced generalization capability to detect defects accurately while minimizing false positives, a balance that is further highlighted by the model's F1-score. The sophisticated data processing scheme that focuses on tokenization process ensures that the data fed into the proposed learning network is of the highest quality clean, structured, and representative of the complex relationships inherent in software code. The findings of this paper have significant implications for software engineering practices, software developers and engineers can expect a significant improvement in the identification and rectification of defects. In future, the scope proposed research work can be extended towards incorporating more advanced and hybrid mechanism and different programming languages towards achieving higher scalability and adaptiveness across other software engineering domains.

REFERENCES

[1] J. Hammond, "Four generations of quality: software and data integrity-an essential partnership?," *Spectroscopy Europe*, Mar. 2022, doi: 10.1255/sew.2022.a7.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016, doi: 10.1109/TSE.2016.2521368.

[3] R. Cummings-John, "The impact poor-quality software can have on businesses," *Forbes*, 2021. Accessed Jan. 19, 2024. [Online]. Available: https://www.forbes.com/councils/forbestechcouncil/2021/01/21/the-impact-poor-quality-software-can-have-on-businesses/

[4] G. Kumar and P. K. Bhatia, "Comparative analysis of software engineering models from traditional to modern methodologies," in *2014 Fourth International Conference on Advanced Computing & Communication Technologies*, Feb. 2014, pp. 189–196, doi: 10.1109/ACCT.2014.73.

[5] B. Dhanalaxmi, G. A. Naidu, and K. Anuradha, "A review on software fault detection and prevention mechanism in software development activities," *Journal of Computer Engineering*, vol. 17, no. 6, pp. 25–30, 2015.

[6] M. A. Haque and N. Ahmad, "An effective software reliability growth model," *Safety and Reliability*, vol. 40, no. 4, pp. 209–220, Oct. 2021, doi: 10.1080/09617353.2021.1921547.

[7] F. Febrero, C. Calero, and M. Á. Moraga, "Software reliability modeling based on ISO/IEC SQuaRE," *Information and Software Technology*, vol. 70, pp. 18–29, 2016, doi: 10.1016/j.infsof.2015.09.006.

[8] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, no. 4, 2020.

[9] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software bug prediction using machine learning approach," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 2, 2018, doi: 10.14569/IJACSA.2018.090212.

[10] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 51–60, doi: 10.1109/MSR.2009.5069480.

[11] H. S. Shukla, "A review on software defect prediction," *International Journal of Advanced Research in Computer Engineering & Technology*, vol. 4, no. 12, pp. 4387–4394, 2015.

[12] S. Pradhan, V. Nanniyur, and P. K. Vissapragada, "On the defect prediction for large scale software systems-from defect density to machine learning," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2020, pp. 374–381, doi: 10.1109/QRS51102.2020.00056.

[13] S. Stradowski and L. Madeyski, "Industrial applications of software defect prediction using machine learning: a business-driven systematic literature review," *Information and Software Technology*, vol. 159, 2023, doi: 10.1016/j.infsof.2023.107192.

[14] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: an empirical study," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 4, pp. 419–424, May 2020, doi: 10.1016/j.jksuci.2019.03.006.

[15] O. Al Qasem, M. Akour, and M. Alenezi, "The influence of deep learning algorithms factors in software fault prediction," *IEEE Access*, vol. 8, pp. 63945–63960, 2020, doi: 10.1109/ACCESS.2020.2985290.

[16] K. K. Bejjanki, J. Gyani, and N. Gugulothu, "Class imbalance reduction (CIR): a novel approach to software defect prediction in the presence of class imbalance," *Symmetry*, vol. 12, no. 3, Mar. 2020, doi: 10.3390/sym12030407.

[17] S. Feng *et al.*, "COSTE: complexity-based oversampling technique to alleviate the class imbalance problem in software defect prediction," *Information and Software Technology*, vol. 129, Jan. 2021, doi: 10.1016/j.infsof.2020.106432.

[18] S. Goyal, "Handling class-imbalance with KNN (neighbourhood) under-sampling for software defect prediction," *Artificial Intelligence Review*, vol. 55, no. 3, pp. 2023–2064, Mar. 2022, doi: 10.1007/s10462-021-10044-w.

[19] H. Turabieh, M. Mafarja, and X. Li, "Iterated feature selection algorithms with layered recurrent neural network for software fault prediction," *Expert Systems with Applications*, vol. 122, pp. 27–42, May 2019, doi: 10.1016/j.eswa.2018.12.033.

[20] H. Alsghaier and M. Akour, "Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier," *Software: Practice and Experience*, vol. 50, no. 4, pp. 407–427, Apr. 2020, doi: 10.1002/spe.2784.

[21] I. Tumar, Y. Hassouneh, H. Turabieh, and T. Thaher, "Enhanced binary moth flame optimization as a feature selection algorithm to predict software fault prediction," *IEEE Access*, vol. 8, pp. 8041–8055, 2020, doi: 10.1109/ACCESS.2020.2964321.

[22] H. Das, S. Prajapati, M. K. Gourisaria, R. M. Pattanayak, A. Alameen, and M. Kolhar, "Feature selection using golden jackal optimization for software fault prediction," *Mathematics*, vol. 11, no. 11, May 2023, doi: 10.3390/math11112438.

[23] A. O. Balogun *et al.*, "Impact of feature selection methods on the predictive performance of software defect prediction models: an extensive empirical study," *Symmetry*, vol. 12, no. 7, Jul. 2020, doi: 10.3390/sym12071147.

[24] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: a semantic LSTM model for software defect prediction," *IEEE Access*, vol. 7, pp. 83812–83824, 2019, doi: 10.1109/ACCESS.2019.2925313.

[25] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Scientific Programming*, vol. 2019, Apr. 2019, doi: 10.1155/2019/6230953.

[26] M. Mafarja *et al.*, "Classification framework for faulty-software using enhanced exploratory whale optimizer-based feature selection scheme and random forest ensemble learning," *Applied Intelligence*, vol. 53, no. 15, pp. 1–43, 2023, doi: 10.1007/s10489-022-04427-x.

[27] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: a framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2022, doi: 10.1109/TDSC.2021.3051525.

[28] X. Yu *et al.*, "Improving effort-aware defect prediction by directly learning to rank software modules," *Information and Software Technology*, vol. 165, 2024, doi: 10.1016/j.infsof.2023.107250.

[29] A. Majd, M. Vahidi-Asl, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "Code4Bench: a multidimensional benchmark of codeforces data for different program analysis techniques," *Journal of Computer Languages*, vol. 53, pp. 38–52, 2019, doi: 10.1016/j.cola.2019.03.006.

[30] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, "Attention based GRU-LSTM for software defect prediction," *PLoS ONE*, vol. 16, no. 3 March, 2021, doi: 10.1371/journal.pone.0247444.

BIOGRAPHIES OF AUTHORS

Deepti Rai completed her master degree in 2016 and bachelor degree in 2004 from Visvesvaraya Technological University, India. She is currently pursuing her doctoral degree in the domain of machine learning at the Department of Computer Science and Engineering, Ramaiah University of Applied Sciences, Ramaiah Technology Campus, Bengaluru, Karnataka, India. She has 10 years of experience in teaching and 6 years of industry experience. Her research interest is in the field of machine learning, deep learning, AI, and cloud computing. She can be contacted at email: deeraisecond@gmail.com.

Jyothi Arcot Prashant completed her Ph.D. in 2020, master degree in 2009, bachelor degree in 2002 from Visvesvaraya Technological University, India. She is currently working as a faculty in the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Ramaiah University of Applied Sciences, Ramaiah Technology Campus, Bengaluru. She has nearly 16 years of experience in teaching and has published many research papers in journals indexed in SCI/SCIE, WoS, Scopus and presented papers in several national and international conferences. Her research interest is in the field of wireless sensor network, MANET, IoT, AI, ML, and deep learning. She is a member of LMISTE, LMIAENG, AMIETE, IFERP, LMINSC, and IEEE. She is a reviewer for Springer, Wiley, IEEE, Elsevier, IGI, and oriental journals and conferences, she is also reviewer of international conference papers from Taiwan, Prague, Czech Republic, and Japan. She has authored book chapter in Springer, Wiley, IET, CRC Press Talyor, and Francis group which are published. She has received best women researcher award and several research excellence awards. She has several Indian patents published; an international patent granted. She can be contacted at email: jyothiarcotprashant@gmail.com.