Journal of Intelligent Systems and Internet of Things
Vol. 18, No. 01, PP. 260-273, 2026
DOI: https://doi.org/10.54216/JISIoT.180120

Design and Construction of the Word Embedding Model for Automated Bug Detection Using Deep Learning Techniques

Khasimbee Shaik[1,*], K .V. Satyanarayana[2], Tirimula Rao Benala[3]

[1]Research Scholar, Department of Computer Science and Engineering, Jawaharlal Nehru Technological University Gurajada Vizianagaram, Dwarapudi, Vizianagaram, Andhra Pradesh-535003, India

[2]Department of CSE (AI&ML), Raghu Engineering College, Visakhapatnam, Andhra Pradesh-531162, India

[3]Department of Information Technology, JNTU-GV College of Engineering, Vizianagaram, Jawaharlal Nehru Technological University Gurajada Vizianagaram, Dwarapudi, Vizianagaram, Andhra Pradesh-535003, India

Emails: khasimbee.shaik786@gmail.com; vsatyanarayana.kalahasthi@gmail.com; b.tirimula@gmail.com

Abstract

Software quality assurance teams can increase productivity and efficiency by expediting the issue-fixing process through automatic localization of bug files. Although source code and bug reports provide valuable semantic information, current bug localization techniques typically underuse it. Numerous deep learning and word embedding models have been developed over time. The word-embedding model used to represent bug reports and the deep learning model used for categorization determine how effective those methods are. Aim of this research is to construct word-embedding method, which has been automated for bug detection using deep learning techniques. Here the input data has been collected as software design based monitored data and processed. Then this data has been analyzed using Bi-LSTM voting vector word embedding model and the feature classification is carried out using convolutional naïve bays attention perceptron neural network in bug detection model. The experimental analysis is carried out in terms of training accuracy, precision, Mean square error, F-1 score, and recall. Furthermore, cross-training datasets from the same and distinct domains are used to gauge how effective the suggested approach is. For datasets in the same domain, suggested system obtains a good high accuracy rate; for datasets in separate domains, it achieves a poor accuracy rate.

Received: March 11, 2025 Revised: June 04, 2025 Accepted: July 09, 2025

Keywords: Word embedding model; Bug detection; Deep learning techniques; Bi-LSTM voting vector; Convolutional naïve bays

1. Introduction

Software flaws that are used by cybercriminals could seriously harm any organization's services and operations. Both static and dynamic approaches to software code analysis have been thoroughly researched by experts and researchers in an effort to enhance software quality. Typically derived from rule-based solutions, dynamic analyzers look at the program control flow. The amount of known vulnerabilities can only be addressed by this method, and upgrading rules calls for specialized knowledge. However, without running the program's source code, static code analyzers find the vulnerabilities [1]. More attention has recently been paid to this topic when it was discovered that software source code and natural language texts have many of the same traits. In particular, research has been done on the automatic detection of code vulnerabilities using NLP applications. Research indicates that deep learning has more potential for

use in static code analysis as it continues to gain traction in other domains, including NLP. For any ML or DL method to produce model inputs as vector representations, a certain embedding technique would be needed. Nevertheless, there are numerous embedding methods in use in field of NLP, including Word2Vec and GloVe [2]. It can be difficult to choose an embedding technique that works with a certain neural model because there are more and more options available. Reporting bugs is a crucial step in testing, development, maintenance of software. Tester or user uses the bug tracking system (BST) in todays agile to register any bugs that are found while using a specific piece of software. A bug report is fed into the BTS. Additionally, it maintains the bug report master list. Typically, bug reports are written in normal language. Users, developers, and testers can all write different versions of the same issue report. The reason for this is that different testers, users, and developers have different vocabularies depending on their level of technical expertise. The triager, a specialist with in-depth expertise of the product, next analyses the substance of the bug report. There are two primary duties for the triager [3]. In order to better comprehend the developer, the triager first rewords the bug report into more technical terms. The second step is for the triager to examine the bug repository for possible duplicate bugs with similar signatures. Additionally, the new bug report is added to master bug report list if it is not a duplicate; if it is, it is regarded as such. However, it takes a great deal of manual labor, time, and thorough bug expertise to filter duplicate issue reports. According to a previous study, over 220 bug reports are sent to BST every day, with 25–30% of those reports being duplicates [4].When duplicate bug reports are given to separate developers for resolution, the developers' time and effort are wasted. Thus, it is highly beneficial to automate process of detecting duplicate bug reports. It saves time, effort, and human costs. The productivity of the developer and triager is also increased by the decrease in physical labor. There are various uses for bug reports. Software testers to document errors and malfunctions use Bug reports, while developers consult them to address issues [5]. In order to gain information, Open Source Software (OSS) projects also gather bug reports in repositories. Description of a bug report does not ensure that the associated problem will be fixed. The characteristics of the original report and the existence of comments are factors that affect how long it takes to repair bugs. The bug report's description and discussion might be another element; however, their effects are not well examined. Prior research has used bug reports to assess effective issue fixes. Predicting, assigning, and searching bug reports have been the goals of some studies. One study looked into the location of bugs. Deep learning has recently been used to identify, forecast, and track bug reports, among other bug-related tasks. Many deep learning predictions, however, still lack a clear foundation [6].

Novelty of the research: This study looked into how a bug report's description and discussion affected how long it took to fix the issue. To build an automated word-embedding model for bug identification through deep learning methods. In this case, the input data was gathered as monitored data based on software design and processed. Convolutional naïve bays attention perceptron neural networks in bug detection models are used for feature categorization once this data has been processed using the Bi-LSTM voting vector word embedding model.

2. Literature Survey

One may classify the bug repair time forecast work as a text classification task. A typical text classification model in contemporary machine learning looks like this: After a set of training text samples that have already been assigned a class are provided, knowledge pertaining to target class is either automatically retrieved by computer algorithms or manually specified by human specialists. This knowledge is then applied to the classification of fresh data samples. Using bug report attributes, Work [7] used Decision Tree analysis to anticipate as well as categories bug reports into two classes: rapid and slow. The author [8] utilized a decision tree-based approach, RF, to classify a defect based on its report attributes into one of three classes—fixed in less than three months, one year, or three years. Work [9], which employed k-Nearest Neighbour (kNN) to forecast whether the fix for a given bug report will be fast or slow using bug report features like severity as well as priority, served as inspiration for applying ML techniques for bug fixing time prediction. Work [10] proposed an approach for finding bug reports with projected bug fixing periods that uses Hidden Markov Models (HMMs) as well as temporal sequences of developer actions. By examining log streams of issue-related activities and text data gathered from bug tracking systems utilising DNN, such as Residual Long Short-Term Memory (RLSTM) and Bi-direction Long Short-Term Memory (BLSTM) techniques, the author [11] suggested a technique for predicting problem fix timeframes over time. Work [12] compared many well-known machine-learning classifiers, such as RF and SVM, on long-lived bug prediction using Bidirectional Encoder Representations from Transformers (BERT) and Term Frequency–Inverse Document Frequency (TF-IDF)-based feature extraction. Additionally, CNN was used in one study to predict whether a problem was security-related, and in another, CNN was used to estimate severity of a software vulnerability [13]. With an emphasis on bug report text, approach can be competitive in terms of prediction performance, notwithstanding the difficulty of directly comparing them because of the differences in target datasets. Pradel and Sen generated code vectors from the custom contexts based on Abstract Syntax Trees (ASTs) using Word2Vec. Deep learning models were trained using these vectors in order to identify

JavaScript code vulnerabilities [14]. Word2Vec was also used to create vector representations from C/C++ source code as well as to train vulnerability detection models using both Word2Vec representations and data from control flow graphs (CFGs).Henkel used the GloVe model to generate vectors learnt from Abstracted Symbolic Traces of C programs [15] in place of Word2Vec. Additionally, FastText was employed in FastEmbed to predict vulnerabilities using ensemble machine learning models [16]. Word embedding approaches have already been used in a number of vulnerability detection cases; however, because baseline dataset types and machine learning model topologies differed, it was not possible to compare these strategies. Utilizing requirement graph as well as graph of files that were changed throughout the bug-fixing process, the author [17] suggested a technique for localizing bugs. The approach demonstrated a relative improvement when compared to the SimiScore and CollabScore techniques. The BugLocator approach for identifying files that caused bugs was proposed by Work [18]. Using this strategy, they built a graph with the new bug as its root, a bug that was comparable to root bug at first level, graph of files that were changed in order to cure level 1 bugs at the second level. The rVMS, a modified version of VMS, is used to calculate the similarity between the bugs. A ranked list of potential files for finding bugs is generated based on this graph. The resemblance between program source files and the bug is the basis for another list. These two lists are combined to form the final list. Tested on 3,000 open source project problems, the suggested approach outperformed state-of-the-art techniques in terms of performance. [19] carried out a thorough empirical investigation. They discovered that the majority of unpleasant odours start when a file is created. However, there are instances where the file starts to smell foul after a number of updates, particularly with Blob and Complex Class. In this instance, the odorous files show a particular inclination towards quality metrics that are entirely distinct from those of clean files. When introducing a new feature or upgrading to an existing one, developers typically generate scents. The efficiency of DL as well as representation learning in bug assignment has been shown in previous research. Notably, bug assignment has rarely made use of the potential and newly popular representation learning techniques such as global vectors (GloVe), embedding is from language models (ELMo), BERT. Similarly, DL methods like LSTM as well as MLP, which are commonly employed in natural language processing, have hardly ever been used for bug assignment. Performance differences may arise from bug assignment techniques based on various word embedding and DL methods [20-22].

3. Methodology

The BugWEmbedNet (bug detection word embedding network) workflow is shown in Figure 1. During the training phase, two distinct word-embedding methods (BiLSTM and voting vector word embedding) are used to first convert bug reports as well as source files into vectors. After that, CNBAPNN is fed matching vectors in order to identify their features. Lastly, the attributes of the source files and problem reports, together with the associated labels (buggy or not) and issue-fixing histories, are paralleled as the input to the improved CNBAPNN. Offline instruction is used throughout the training phase. Trained method will effectively use bug reports to find locations of buggy files whenever receive new bug reports.

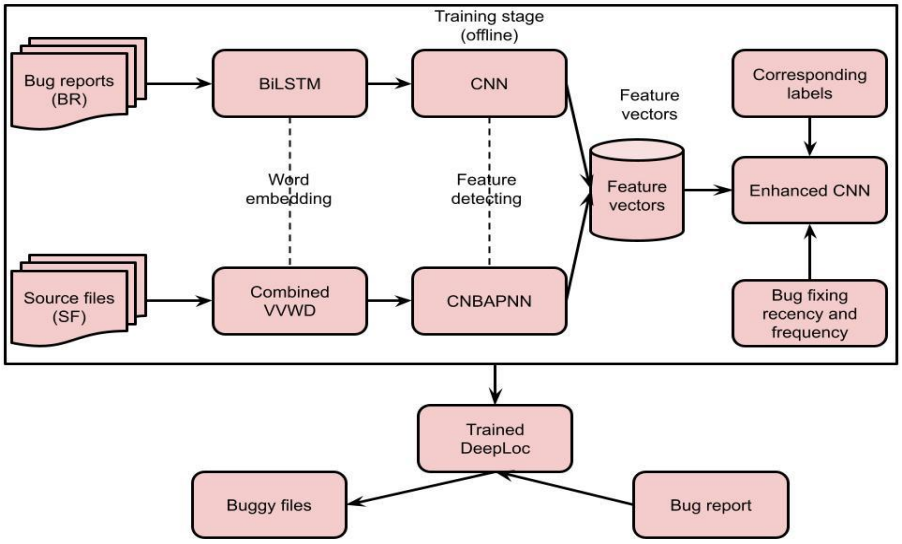


Figure 1. Proposed BugWEmbedNet architecture

Method first loaded dataset files as well as converted them into word token sequences, which were initially variables, code identifiers, etc. Each of these sequences represents a semantic function representation. The list of vulnerability labels was produced by parsing the file names. The encoding phase comes next. In this case, the system trained and mapped the token sequences into vector representations using the established word embedding approach. After that, these vectors will be divided into test and training sets. To train the vulnerability detector, a specific neural network receives 80% of the vector representations. The trained detector tested the remaining representations when the training process was complete. The susceptibility probabilities for the test samples were generated and saved to a raw CSV file following the model's evaluation. In order to handle the data from the CSV file and organize the list of test names into a different table according to the likelihood of their vulnerabilities, this system lastly executed a script. After that, the detection logs and performance metrics calculations were automatically collected.

3.1 Bi-LSTM voting vector word embedding (BiLSTM_VVWD) model

This module into a vector that preserves the original code's syntactic and semantic information transforms each code function file. The information must be extracted from the code tokens in order for the models to learn efficiently later on. Specifically employed the embedding layer to translate these identifiers in code to the semantic vector representation in order to preserve the semantic knowledge conveyed by the identifier names. Over 90% of the sequences in the two datasets were found to have lengths less than 1000 by looking at their content. Set the maximum code sequence length at 1000 in order to balance sequence length with sparsity. Functions that contain a sequence length more than 1000 are trimmed to that length. On the other hand, for sequences less than 1000, insert zeros at the end. To increase the model's accuracy and address the problem of class imbalance, which could have a detrimental impact on method output, prepare data before developing method.

Integer vectors cannot carry abstract Syntax Tree (AST) token context information. Consequently, every integer vector is converted to a dense vector using word embedding technique. Every word in a vector representing an input sequence is represented by a big sparse vector in classical embedding's. However, due to the large size of the input sequences, this representation is sparse. For instance, AST tokens will remain intact, but their relationship will be disregarded. Rather, dense vectors of a defined length are utilized to represent words in word embedding, where a vector is the projection of a word into a continuous vector space. For instance, dense vector [0.25,0.1] has a "For Statement" node embedded in it. There are two primary advantages of word embedding. First off, compared to the sparse vector, embedded vector has smaller dimensions. Second, in vector space, AST nodes with the same context are situated close to one another. The data's sequential order is maintained. By correctly capturing essential traits, Bi-LSTM implementation aids in the detection of long-term dependency information.

Several LSTM cells are connected to build the LSTM model. Data Z i is kept in cell structure is determined by input gate. By controlling the cell information, each gate completes forgetting as well as remembering duties. Following is precise evaluation formula (1):

$$\tilde{c}^t = \tanh(W_c[h^{t-1}, x_t] + b_c,$$

$$Z^i = \sigma(W_i[h^{t-1}, x_t] + b_i)$$

$$Z^f = \sigma\left(W_f[h^{t-1}, x_t] + b_f\right)$$

$$Z^o = \sigma(W_o[h^{t-1}, x_t] + b_o)$$

$$c^t = Z^i \times \tilde{c}^t + Z^f \times c^{t-1}$$

$$h^t = Z^o \times \tanh c^t$$

$$y^t = \sigma\left(W_y \times h^t\right) \quad (1)$$

Weights are Wc, Wi, Wf, and Wo; bias vectors are bc, bi, bf, and bo; the activation function is σ; transfer of the prior state and the current input are denoted by h t−1 and xt. Figure 2 depicts the data processing procedure for one Bi-LSTM layer.

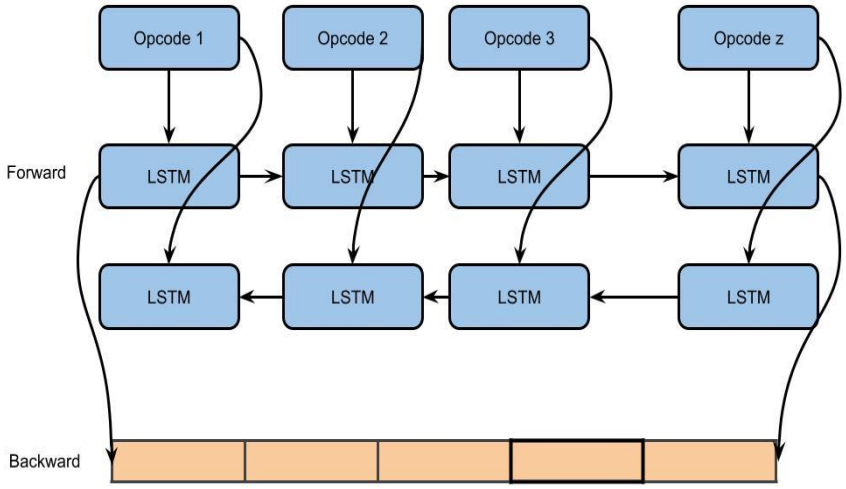


Figure 2. Bi-LSTM data processing process.

The attention mechanism because this method compute time step t based on evaluation result at t − 1. Processing data sequences that are too long will result in a significant decrease in performance. It is essentially a query that leads to a sequence of (key-value) accurate mappings. It saves resources and rapidly obtains the most useful feature information by concentrating limited attention on key opcodes and giving them weights, which is evaluated as shown in Equation (2).

$$H = \tanh(W_a \cdot X_c + b_a)$$

$$\alpha = \text{softmax}(W^T \cdot H)$$

$$Y = X_c \cdot \alpha^T \qquad (2)$$

where ba represents the attention layer's bias vector and Wa represents the weight matrix of the attention layer. Table 1 displays output dimensions as well as characteristics of every network layer. The output of Bi-LSTM and attention layers is still high-dimensional data in comparison to classifiable vulnerability classes. To transform output vector into the dimensions of a multi-label vector, included a fully connected layer.

Table 1: Output and parameters of every network layer

Network Layer	Output Size	Number of Parameters
Bi-LSTM layer	(16,6290,256)	2,021,376
Fully connected layer	(16,5)	645
Attention layer	(16,128)	16,521

Bug reports are composed of sentences and written in natural languages, so rather than turning every word into a vector, turn each sentence into a vector. Condensed and abridged information is always written in a single sentence for summaries. Important information is hard to remember if summaries are treated like sentences. However, descriptions typically have many phrases and some material that is repeated. To use the following equation to determine vectors of every line in the source code by eqn (3)

$$V_l = \frac{1}{n_l}\sum_{i=1}^{n_l} w_i v_i = \frac{1}{n_l} wv \qquad (3)$$

To speed up computing process, it is preferable to adopt appropriate victories form during implementation.

3.2 Convolutional naïve bays attention perceptron neural network (CNBAPNN) in bug detection model

Figure 2 shows main layout of bug triage system. Preprocessing stage is applied to the data. The word embedding technique is then used to transform processed data into word vectors. CNN network uses these word vectors as input. The CNN model predicts a ranked list of suitable developers after training on provided word vectors. The convolution layer uses convolutional filters c k to perform a convolution of the input matrix z, with a distinct output calculated for each filter. Features of varying lengths are extracted from the input vector using three distinct filters with kernel sizes k = 3, k = 4, and k = 5. The word vector dimensions are fixed at 300, thus zero padding (P) is utilized whenever needed. The convolution operation is applied to n-length data to extract feature map F = {f1, f2, f3,... fl.., f(n−k+1)}. fl stands for lth feature in F. Each of three convolution filters has the following shapes: in channels, out channels, height, and breadth. Three convolution filter types are employed with widths of 300 and heights of 3, 4, and 5. Each layer contains 256 filters or neurons, resulting in 1 in-channel and 256 out-channels.

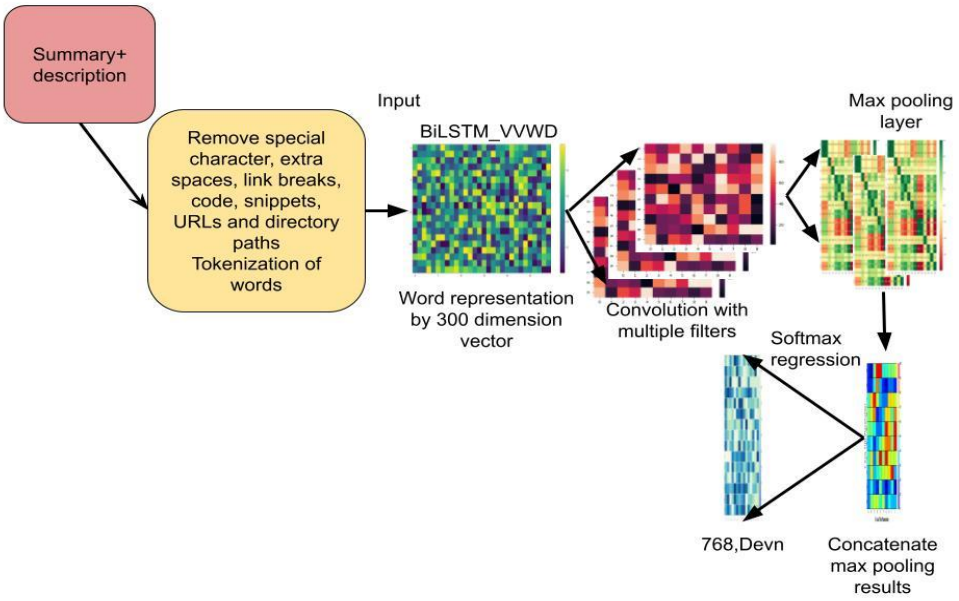


Figure 3. Architecture of CNBAPNN

A number of models make different assumptions about Naïve Bayes fitting. The most popular models are: There are two types of Bernoulli event models: the multinomial model, which employs feature occurrence frequencies, Boolean weight model, which uses binary feature occurrences. Let C = {C1,C2…….,Cn} be bug classification into n distinct classes. In order to classify the invisible insect (Bi) with a greater posterior probability, (2) will be used by eqn (4)

$$\mathrm{P(Ck, Bi)} = \mathrm{P(Bi \mid Ck)(P(Ck)/P(Bi)}$$

$$\mathrm{P(Ck)} = \mathrm{Nk/N} \qquad (4)$$

How is P(Bi|Ck) estimated? Instead of using word data as input, use feature data for features related to bugs and word information for features of the natural language type. Surface lexical and semantic similarity are then represented by greatest value of bug report similarity with all models as well as entire document.

3.3 Attention Mechanism

Hidden characteristics of every time node in a sequence can be obtained from the Bi-LSTM network's output. These nodes make different contributions to representation of sequence meaning. After Bi-LSTM Layer, embed an attention layer to increase the impact of crucial nodes. A sequence vector is created by combining crucial nodes that are important to the sequence's meaning using the attention mechanism. The complete procedure is depicted in Figure 4, explain it as follows (5):

$$u_{it} = \tanh(W_n h_{it} + b_n)$$

$$\alpha_{it} = \frac{\exp(u_{it}^{\top} u_n)}{\sum_t \exp(u_{it}^{\top} u_n)}$$

$$s_i = \sum_t \alpha_{it} h_{it} \quad (5)$$

Next, use a softmax function to get a normalized importance weight αit and quantify node's relevance as dot product similarity of uit with un. The sequence vector si is then determined by adding up all of the nodes with the appropriate weights. During training, the node level context vector un can be updated and is initialized at random.

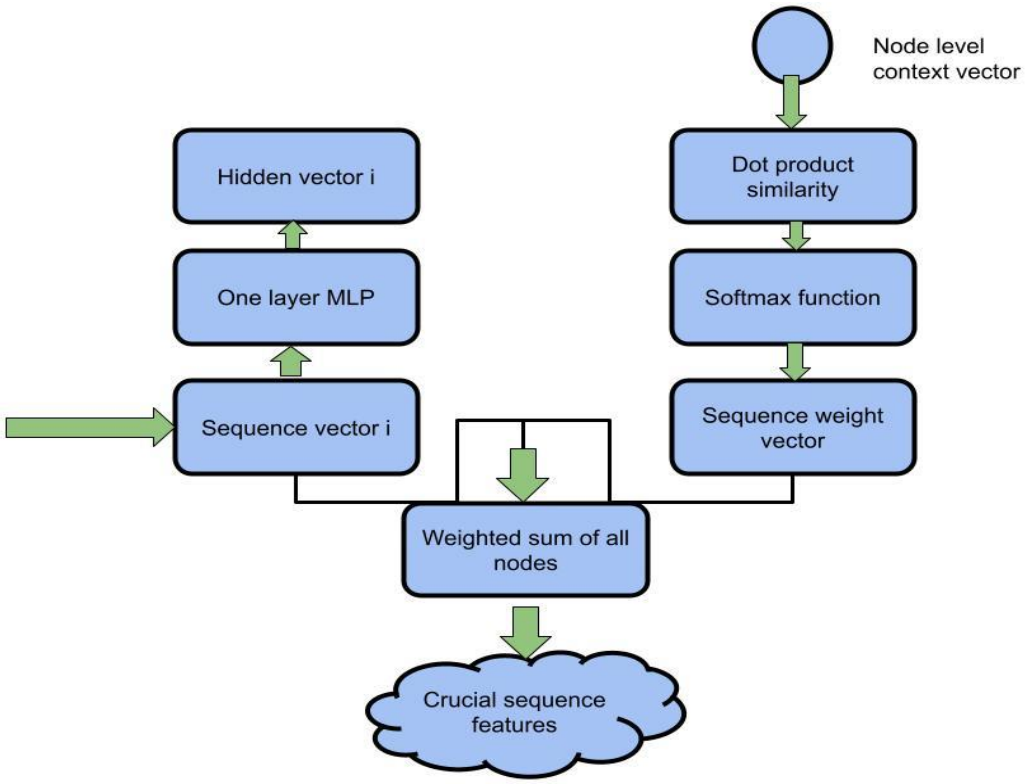


Figure 4. The process of attention mechanism

Let xE be the model's input vector, z k i be output of neurone i in layer lk, and w k ij represent weight of connection between ith neurone of lk layer and j th neurone (if k ≥ 1) of the lk−1 layer. For every neurone, add an additional weight parameter, with rk denoting number of neurones in layer lk and b k i denoting bias for i neurone in layer. A feed-forward phase is used to obtain output z from MLP methoed. The input layer l0 is initialised first, values of neurones' z 0 i outputs are set to their corresponding inputs in the vector xE = {x1,..., xn}. From l1 to lm−1, compute the weight sums and outputs for each hidden layer using Formula (6).

$$s_i^k = \overrightarrow{w_i^k} \cdot \overrightarrow{z^{k-1}} + b_i^k$$

$$s_i^k = b_i^k + \sum_{j=1}^{r_{2-1}} w_j^k \cdot z_j^{k-1} \text{ for } i = 1, \ldots, r_k$$

$$z_i^k = f_n(s_i^k) = \begin{cases} s_i^k, & s_i^k > 0 \\ 0, & \text{else} \end{cases} \text{ for } i = 1, \ldots, r_k \quad (6)$$

$$s_1^m = \overrightarrow{w_1^m} \cdot z^{m-1} + b_1^m = b_1^m + \sum_{j=1}^{m-1} w_{j1}^k \cdot z_j^{k-1}$$

$$\hat{y} = z_1^m = f_z(s_1^m) = \frac{1}{1+e^{-x_1^m}} \quad (7)$$

The learning process relies on iteratively modifying values of −→ w k i and b k i in order to minimise L loss function, where fz activation function for output layer is a logistic function (sigmoid). However, in this instance, the loss function is CrossEntropy, which is a logarithmic function and is expressed as Formula (8).

$$J = L(\hat{y}_i, y_i) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \quad (8)$$

where yi represents real value for the input-output pair −→ xi, yi, and yˆi represents the MLP model's estimated output on input −→ xi. The objective is to use the gradient descent technique to minimize J with regard to all w k ij and b k i. As a result, the Adam optimizer is utilized to modify specifications w k ij and b k i with α as a learning rate.

Algorithm for BugWEmbedNet

S-IP ← Source IP
if S − IP == GET _request or S − IP == POST _request then
ProxyIP-count – start
Go to step 7
else Go to step 1
Packet_lenght [] = SourceIP_paketlength
Se_Time [] = SourceIP_Se_Time
if S-P == Anonymous_ProxyList-IP then
elseGo to step 13
ProxylP-count++
if S_IP-Port==Constant then
elseSourcePort == Varying
SourcePort == Constant
for i in range (0, Se _Time[ ]) do
if arr[i] == arr[] then
for$j$ in range ($i + 1$, Se_Time[ ]) do
No of host equal Session Time ++
for j in range ($i + 1$, Packet_length[1) do
fori in range (0 , Time_Frame[ ]) do
if arri] == arr] then
end if
No of host equal Packet frame Size ++
if No of host equal Session Time ≥ 20 then
TQ == High
PX − IP == High
ifProxylP-count ≥ 20 then
elsePX-IP == Low
end for

4. Experimental analysis

Simulation setup- The LinuxMint-19-tara operating system, Kernel 4.15.0-42-generic, 16 GB of RAM, Intel Xeon CPU E-5-2620 v3 @ 2.40GHz, and GPU NIVIDA (Quadro K220) were used during entire experiment. Tensorflow (1.12.0) is compatible with Keras version 2.2.4 and the Python framework version 3.6.7.

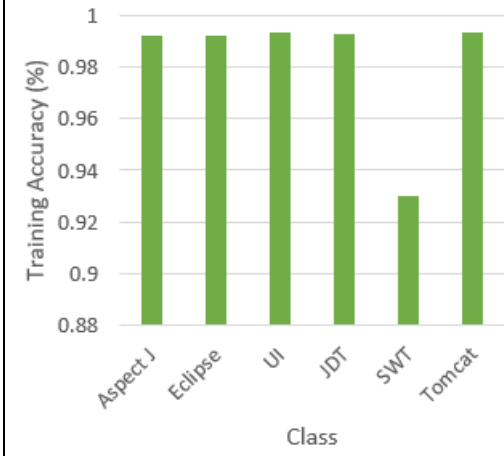
Not able to locate original mapped source files for an old bug without before-fix versions since some source files might have been altered or even removed. Therefore, using publicly accessible mappings of bug reports as well as accompanying commit history, link each bug to a dataset (Table 2) that contains source file versions prior to fixes.

Table 2: Subject projects

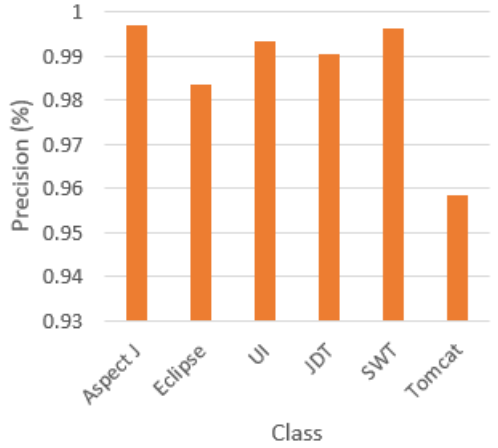
Project	**Time Range**	**# of Bug Reports**	**# of Fixed Buggy Files**	**Avg. # of Buggy Files per Bug**	**Max. # of Words per Bug Report**	**Avg. # of Words per Bug Report**
AspectJ	03/02 – 01/14	593	1,151	4.0	4,068	188.5
SWT	02/02 – 01/14	4,151	1,415	2.1	5,125	125.0
JDT	10/01 – 01/14	6,274	5,002	2.6	5,233	137.4
Eclipse UI	10,01 – 01/14	6,495	6,228	2.7	4,468	124.6
Tomcat	07/02 – 01/14	1,056	1,038	2.4	2,320	87.5

Each project's chronologically arranged bug reports are divided into three groups: the training set, which consists of oldest defects, validation/development set, which consists of validation/development set, test set, which consists of newest bugs. For a particular bug report, there are many source files, and due to the required training time and memory, it is not possible to include them all in the training set. All source files are matched with every issue report in the validation and testing collection.

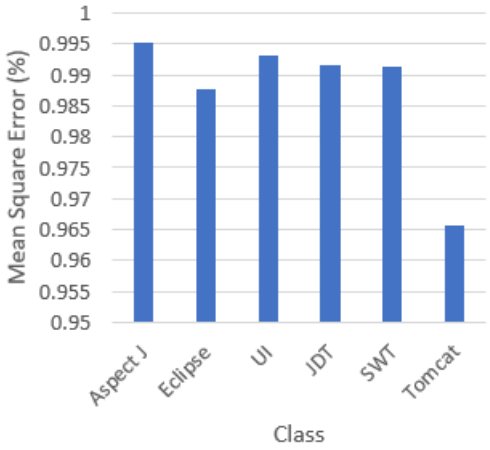
A total of 138,569 samples make up final dataset, which was produced in a homogeneous and randomized manner. Of these, 100,000 are benign samples, while 38,569 are malevolent samples with 41 properties. The final method evaluation should preferably be performed on a test dataset that are utilized for parameter tuning or method training in order to offer an unbiased evaluation of method effectiveness. Predicted score may only be based on a single validation set as well as unlikely to accurately reflect method overall performance. Dataset was randomly partitioned into 2 halves with an 80%:20% partition ratio by dynamic feature extractor model. The complete report produced using confusion matrix for evaluating method performance is displayed in Table 3, performance analysis is displayed in terms of training accuracy, precision, mean square error, F-1 score, recall, and AUC in Figure. 5(a)–(f).



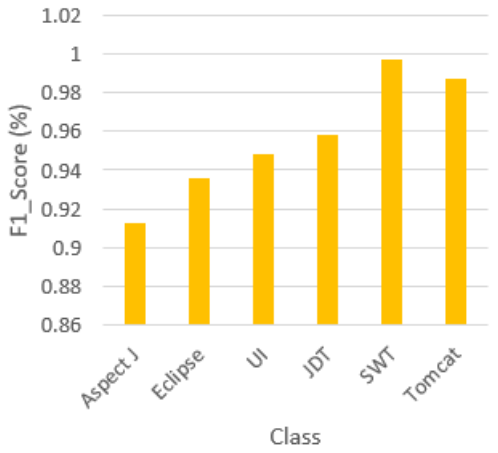
training accuracy



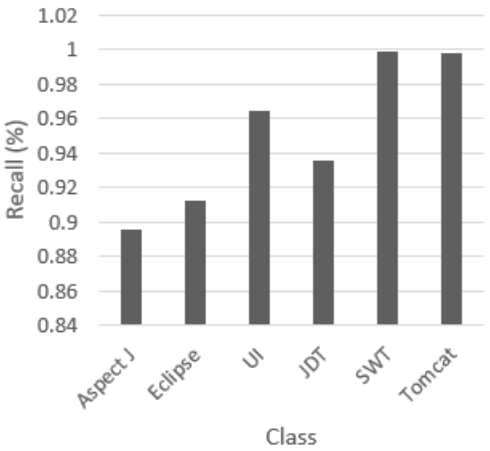
precision



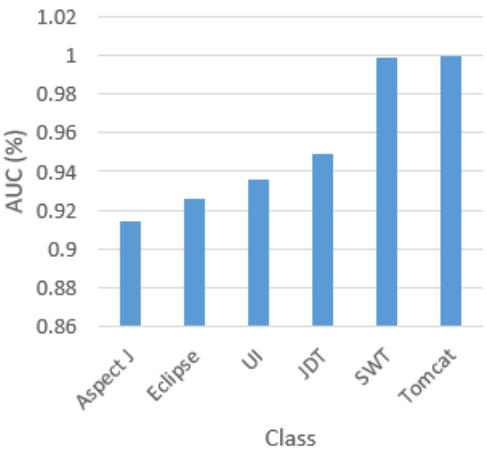
Mean square error



F-1 score



recall



AUC

Figure. 5(a)–(f) Training accuracy, precision, mean square error, F-1 score, recall, and AUC

Table 3: Experimental Results on Test Dataset.

Class	Training accuracy	Precision	Mean square error	F1_Score	Recall	AUC
Aspect J	0.9924	0.9969	0.9953	0.9125	0.8952	0.9145
Eclipse	0.9921	0.9835	0.9877	0.9356	0.9125	0.9258
UI	0.9932	0.9932	0.9932	0.9486	0.9645	0.9356
JDT	0.9928	0.9902	0.9915	0.9586	0.9358	0.9487
SWT	0.9298	0.9961	0.9914	0.9975	0.9987	0.9986
Tomcat	0.9935	0.9585	0.9658	0.9875	0.9981	0.9995

Table 4: Results of inspecting and classifying warnings in real-world code

Bug detector	Reported	Quality problem	Bugs code	False positive
Swapped arguments	50	0	23	27
Wrong binary operand	50	0	35	15
Wrong binary operator	50	7	37	6
Total	150	7	95	48

Train every bug detector using 100,000 training files, then apply trained bug detector to 50,000 validation files, lastly examine code places that bug detectors flag as possibly erroneous in order to examine how well Bugs performs on real-world code. Assign each warning to one of three groups based on the inspection: • A bug. If code is flawed in sense that it does not produce desired runtime behavior, a warning indicates a bug. • Issue with code quality. If code produces the intended runtime behavior but still needs to be modified to be more efficient or less prone to errors, this is a warning sign of a code quality issue. Code that deviates from generally recognized standards and programming guidelines that are typically examined by static lining tools falls into this category. Because issues with code quality frequently correlate with odd code that deviates from a standard coding idiom, learnt bug detectors identify these issues. • A false positive. Otherwise, a caution is a false positive. Cautiously consider a code position to be a false positive and uncertain about its intended behavior. Additionally, came across a number of code samples with deceptive identifier names, which categories as false positives due to subjective nature of determining if an identifier is deceptive. The findings of the inspection and classification of warnings are compiled in Table 4. Sixty-eight percent of the 150 warnings that were examined indicated a real issue, with 95 pointing to bugs and 7 to a code quality issue. Current manual bug detectors mostly use heuristics to filter likely false positives, but they usually offer comparable true positive rates. Conclude that this learnt name-based bug detectors provide excellent precision and are successful in detecting real-world issues.

However, dataset imbalance between vulnerabilities as well as non-vulnerabilities demonstrated that these metrics will undervalue method detection performance in many vulnerability detection scenarios. The ranking retrieval precision and recall are the measures used to assess these classifiers. Furthermore, since the retrieval job of vulnerable functions was the focus of this technique, these metrics are highly suggested for this work and would be more suitable

for assessing the detection outcome. Final column also indicates the proportion of samples that must be referred to designer. In ten initiatives, this percentage averages 36.7. It is evident that the data is unbalanced and that there are significantly fewer samples with label 1 than there are with label 0. The usage of imbalanced data biases machine-learning algorithms towards more frequent classes, which ultimately results in model overfitting, because these algorithms were created and optimized to train with balanced data. Data must be balanced in order to resolve this issue. There are two methods for balancing data: up sampling and down sampling. To equal amount of samples with various classes, the downsampling method eliminates certain samples that have a class label with a higher frequency. This approach results in less training data and the loss of appropriate training samples. Data with continuous numerical values are subjected to this approach. These characteristics can be applied to the text once it has been converted to a vector. In order to create new samples from them, one sample with a lower class frequency is chosen using this procedure. The K neighbor of each sample is chosen.

The effect point estimate is always included in confidence intervals, which also contain the true population value based on the confidence level, which is often set at 95%. A larger sample size has the effect of lowering the P value, but it merely narrows the width of the confidence interval around the same effect size. Whereas confidence intervals shift the interpretation of the results to the size of the effect and the relationship and its range of plausible values provided by the data under study, the P value restricts the interpretation of the trial results to the binary of significant and no significant. As seen in figure 6, confidence intervals change the interpretation from a qualitative assessment to a quantitative estimate of the effect.

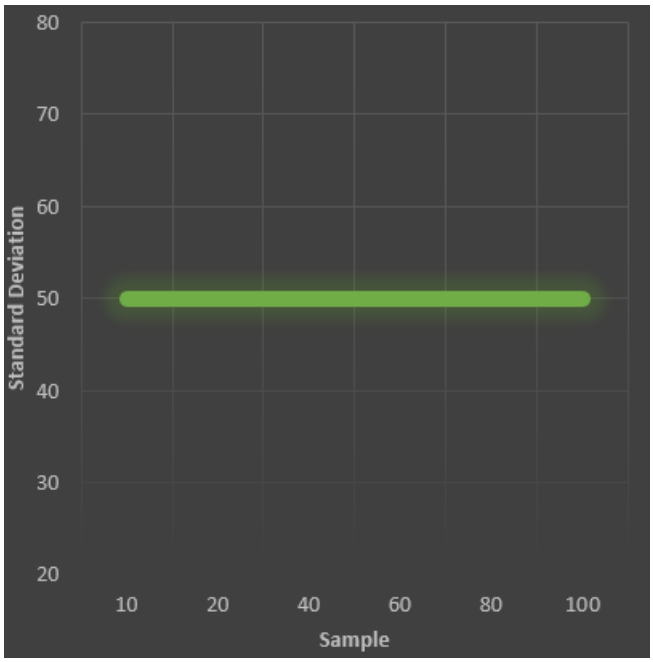


Figure 6. From a target population with a mean of 50 and a standard deviation of 10, 100 samples were selected. Each sample's 95% CI around the true population mean shows that 95 out of 100 confidence intervals drawn contain the true mean.

A random selection is made from among these samples. A new vector that is added to the data set as a new instance is produced by multiplying this vector by a random number between 0 and 1. Until the data is balanced, this process is repeated. The suggested model needs to be tested using the remaining data and trained using a portion of the dataset in order to be evaluated.To assess, employed the K Fold validation method. K. determines percentage of data for testing. For instance, in 10 folds, split the data into 10 parts, choose one for testing, use the remaining 9 parts to train the model. After ten iterations of this procedure, the evaluation's average of the parameters is presented. The model must be tested with various K in order to determine the optimal K. Deep learning-based models require time to train, thus in order to choose the right K, a project with a sufficient amount of deep learning-ready samples is chosen, and models with Ks rated 10, 5, 3, and 2 are assessed. The model is better trained and there are more training examples for larger K; conversely, there are fewer test samples and the stability of the results should be examined. Because both

datasets belong to the same domain and genre—the Integrated Development Environment—results show that the suggested technique likewise predicts with good accuracy. Additionally, Eclipse dataset is utilized to train suggested system, Open Office dataset is utilized for testing. Because the datasets in this cross-training experiment come from diverse domains, its accuracy is low. It is shown that the suggested system learns the latent characteristics of the domain in addition to the generic syntactic and semantic relationships of features, which improves the outcomes of cross-training projects within the same domain.

4.1 Discussion

In this study, problem reports were categorized into two groups based on the time it took to fix them using comments. Experience and bug difficulty were not taken into account. Internal legitimacy is in danger because of this. Furthermore, only bug reports that were closed were used. While some problem reports had only one or two comments, others had no discussion at all. Another risk to internal validity is that if the model learns from bug reports with few comments, the outcomes could be different. The impact of additional criteria, such the level of debate and comment maturity, on feature extraction should be examined in order to resolve this problem. The experimental findings might be unique to this study's dataset. External legitimacy is under danger as a result. Different models may use different extracted features, which could lead to different predictions. In order to find out which DL methods are more suited for bug assignment tasks, formulated this research topic. Other datasets ought to be added in the future to see if comparable outcomes are produced.

4.2 Limitations

The limitations of this work are explained in this section. The majority of automated methods for bug triage that use machine learning techniques, such as this study, usually take advantage of data from previously fixed issues. These methods recommend a group of developers based on their track records of participating in bug-fixing activities; as a result, they do not include new developers who are not listed among the developers who have fixed problems in past. For instance, if a newly recruited developer is not listed as a fixer for any issues that have been fixed, the automatic trigger will not evaluate the developer as a candidate. This constraint also affects this method. Additionally, the normal bug triage dataset is skewed and extremely unbalanced. Many developers have only addressed a small number of problem reports. The machine learner may treat these developers as outliers during training, which could have a detrimental effect on the models' performance.

5. Conclusion

Building an automated word-embedding model for bug identification through deep learning techniques is the goal of this study. In this case, the input data was gathered as monitored data based on software design and processed. Convolutional naïve bays attention perceptron neural networks in bug detection models are used for feature categorization once this data has been processed using Bi-LSTM voting vector word embedding model. To determine how similar bug reports are, DL method uses two layers: the Similarity Measurement layer and the Fully Connected layer. Similarity measurement metric in the Similarity Measurement Layer compares the bug reports' sentence representations. The Fully Connected Layer calculates two bug reports' similarity scores. Based on similarity ratings, bug reports are then categorized as duplicate or non-duplicate. Suggested method suggests a prioritized list of ten suitable developers after learning summaries as well as descriptions from bug reports. As an assessment metric, employ top-k accuracy. According to comparison of experimental findings, increasing the batch size is a good choice if a sizable dataset with a significant number of classes is available. Enhancing number of filters is a good choice if there is a tiny dataset. Intend to test alternative neural network architectures, like a bioinspired spiking CNN (SCNN), for bug triage difficulties in the future. Because of its bio-realism, this network, which belongs to the third generation of neural networks, is thought to perform better than conventional, non-spiking neural networks.

References

[1] Wang, R., Ji, X., Xu, S., Tian, Y., Jiang, S., & Huang, R. (2024). An empirical assessment of different word embedding and deep learning models for bug assignment. *Journal of Systems and Software*, 210, 111961.

[2] Chen, B., Zou, W., Cai, B., Meng, Q., Liu, W., Li, P., & Chen, L. (2024). An empirical study on the potential of word embedding techniques in bug report management tasks. *Empirical Software Engineering*, 29(5), 122.

[3] Sepahvand, R., Akbari, R., Jamasb, B., Hashemi, S., & Boushehrian, O. (2023). Using word embedding and convolution neural network for bug triaging by considering design flaws. *Science of Computer Programming*, 228, 102945.

[4] Devi, M. C., & Rajkumar, T. D. (2025). A novel attention-based deep learning model for software defect prediction with bidirectional word embedding system. *Soft Computing*, 1-18.

[5] Patil, A., Han, K., & Jadon, A. (2024, March). A comparative analysis of text embedding models for bug report semantic similarity. In *2024 11th International Conference on Signal Processing and Integrated Networks (SPIN)* (pp. 262-267). IEEE.

[6] Asudani, D. S., Nagwani, N. K., & Singh, P. (2023). Impact of word embedding models on text analytics in deep learning environment: a review. *Artificial Intelligence Review*, 56(9), 10345-10425.

[7] Isotani, H., Washizaki, H., Fukazawa, Y., Nomoto, T., Ouji, S., & Saito, S. (2021, September). Duplicate bug report detection by using sentence embedding and fine-tuning. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 535-544). IEEE.

[8] Hamdy, A., & Ezzat, G. (2022). Deep mining of open source software bug repositories. *International Journal of Computers and Applications*, 44(7), 614-622.

[9] Manoharan, R. (2025). Improving Security and Performance in Chaotic Optical Communication via Real-Time Pilot Signal Processing Techniques. *IETE Journal of Research*, 1-9.

[10] M. Rajesh, S. Ramachandran, K. Vengatesan, S. S. Dhanabalan, and S. K. Nataraj, "Federated Learning for Personalized Recommendation in Securing Power Traces in Smart Grid Systems," in *IEEE Transactions on Consumer Electronics*, vol. 70, no. 1, pp. 88-95, Feb. 2024, doi: 10.1109/TCE.2024.3368087.

[11] Messaoud, M. B., Miladi, A., Jenhani, I., Mkaouer, M. W., & Ghadhab, L. (2022). Duplicate bug report detection using an attention-based neural language model. *IEEE Transactions on Reliability*, 72(2), 846-858.

[12] Dharmakeerthi, P. G. S. M., Rupasingha, R. A. H. M., & Kumara, B. T. G. S. (2024, February). CNN-Based Deep Learning Approach for Prioritization of Bug Reports. In *2024 4th International Conference on Advanced Research in Computing (ICARC)* (pp. 31-36). IEEE.

[13] K. Ali, M. A. Jamshed, and S. A. Khan, "Deep Learning for Software Bug Detection: A Comprehensive Review," *Journal of Software: Evolution and Process*, vol. 36, no. 2, p. e2295, 2024.

[14] Tang, W., Tang, M., Ban, M., Zhao, Z., & Feng, M. (2023). CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software*, 199, 111623.

[15] Bibyan, R., Anand, S., Jaiswal, A., & Aggarwal, A. G. (2024). Bug severity prediction using LDA and sentiment scores: A CNN approach. *Expert Systems*, 41(7), e13264.

[16] Du, X., Zheng, Z., Xiao, G., Zhou, Z., & Trivedi, K. S. (2021). Deepsim: Deep semantic information-based automatic mandelbug classification. *IEEE Transactions on Reliability*, 71(4), 1540-1554.

[17] Viswanadhapalli, V. (2024). Automated Bug Detection and Resolution Using Deep Learning: A New Paradigm in Software Engineering. *International Journal of Engineering and Computer Science*, 13(04).

[18] Siachos, I., Kanakaris, N., & Karacapilidis, N. (2025). Software bug prediction using graph neural networks and graph-based text representations. *Expert Systems with Applications*, 259, 125290.

[19] Ali, W., Bo, L., Sun, X., Wu, X., Memon, S., Siraj, S., & Ashton, A. S. (2023). Automated software bug localization enabled by meta-heuristic-based convolutional neural network and improved deep neural network. *Expert Systems with Applications*, 232, 120562.

[20] Mula, V. K. C., Kumar, L., Murthy, L. B., & Krishna, A. (2022, September). Software Sentiment Analysis using Deep-learning Approach with Word-Embedding Techniques. In *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)* (pp. 873-882). IEEE.

[21] Zaidi, S. F. A., Woo, H., & Lee, C. G. (2022). Toward an effective bug triage system using transformers to add new developers. *Journal of Sensors*, 2022(1), 4347004.

[22] Mahajan, G., & Chaudhary, N. (2022). Design and development of novel hybrid optimization-based convolutional neural network for software bug localization. *Soft Computing*, 26(24), 13651-13672.