

Article

Software Fault Localization through Aggregation-Based Neural Ranking for Static and Dynamic Features Selection

Abdulaziz Alhumam 

Department of Computer Science, College of Computer Sciences and Information Technology, King Faisal University, Al-Ahsa 31982, Saudi Arabia; aahumam@kfu.edu.sa

Abstract: The automatic localization of software faults plays a critical role in assisting software professionals in fixing problems quickly. Despite various existing models for fault tolerance based on static features, localization is still challenging. By considering the dynamic features, the capabilities of the fault recognition models will be significantly enhanced. The current study proposes a model that effectively ranks static and dynamic parameters through Aggregation-Based Neural Ranking (ABNR). The proposed model includes rank lists produced by self-attention layers using rank aggregation mechanisms to merge them into one aggregated rank list. The rank list would yield the suspicious code statements in descending order of the rank. The performance of ABNR is evaluated against the open-source dataset for fault prediction. ABNR model has exhibited noticeable performance in fault localization. The proposed model is evaluated with other existing models like Ochiai, Fault localization technique based on complex network theory, Tarantula, Jaccard, and software-network centrality measure concerning metrics like assertions evaluated, Wilcoxon signed-rank test, and Top-N.

Keywords: fault localization; neural ranking; parameter selection



Citation: Alhumam, A. Software Fault Localization through Aggregation-Based Neural Ranking for Static and Dynamic Features Selection. *Sensors* **2021**, *21*, 7401. <https://doi.org/10.3390/s21217401>

Academic Editor:
Naveen Chilamkurti

Received: 11 October 2021
Accepted: 3 November 2021
Published: 7 November 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

It is hard for developers to find and correct software flaws while the program is being developed and after its release. The complete process of bug identification is time-consuming, and the automated software does the job perfectly through fault localization. Using fault localization models reduces the debugging expenses and allows the developers to devote more to possible vulnerable components of the software. Traditionally, fault localization, which focuses on locating defects in software manually, has been a difficult, time-consuming, and costly job because of the complexity of large-scale software systems [1]. The developer's knowledge, expertise, and judgment are required to locate and classify the snippet which is most likely to contain bugs. As a result of the constraints mentioned above, interest has been revived in creating automated models for localization of faults in software while at the same time minimizing human involvement.

The automated models are classified as static and dynamic models for fault localization [2]. The static model does consider various static features of the software, such as the log reports of the bugs, bug history reports, code modification logs, and performs the fault matching with the existing issues [3,4]. Moreover, the static models mainly recognize the errors in the code like using the dangling pointers, syntax errors, security-related access privileged errors in snippets, and the code-tempting issues [5]. The dynamic models relay the dynamic characteristics for the bug identification by introspecting the code's response in the run-time environment. The use of test cases would assess the vulnerability severity and rank for each such code snippet in the software. However, in either of the models, the bug localization in developing robust software is quite far from a satisfactory outcome.

Fault localization is a time-consuming and laborious task for dealing with massive software projects for two primary reasons. The formal reason is that there are typically

many pending faults to be still found and fixed. The well-known Eclipse bug repository has reported around 200 issues each day approaching release dates, whereas the Debian project has reported around 150 issues in a single day [6]. Another pivotal reason is the time taken for fault isolation. Gaeul Jeong [7] has observed that almost all problems take 100 to 200 days to fix in the PostgreSQL project. To repair just half of the issues takes nearly 300 days. Most issues in the Tomcat project take 40–200 days to fix. It is assumed that 5% of all problems may take almost two years to fix in the worst-case scenario.

The features are exceptionally important in the fault localization process, either static or dynamic features. When any of those features are considered alone, few challenges will make the process of fault identification challenging. When the static features are considered alone, the assessment model may fail to recognize the dynamic software features. Furthermore, the significant features and essential functional characteristics will be missing when dynamic features alone are considered. The deep neural network models efficiently consider both features with an exceptional learning capability [8]. Various studies on fault localization incorporate something into the software to track the model's functionality and outcome [9,10]. The different fault localization techniques include programming constraints, program log reports, breakpoints, profiling. Moreover, all the fault localization tools are designed by incorporating all these features for the effective debugging of the software.

Program constraints are assertions introduced to a program that must be true to function correctly throughout its execution. These constraints are generally specified in the program as a conditional statement, which causes the program to stop execution if any constraint is not satisfied [11]. Program log [12] inserts are often included with the code on a hit-or-miss basis to check variables and other program information updating processes that utilize stored log files or printed run-time information to identify failure when they see strange program behavior. Using the breakpoints [13] is the other approach used in fault localization. It will halt the program whenever it approaches a pre-set spot, enabling the programmer to check current conditions. The programmer may alter the code and resume the program's execution to monitor the response of the code with a possible bug. The run-time profiling [14] of the program performance indicators such as execution speed and memory consumption is known as profiling. It is often used to validate the code for the issues like memory leaks, unexpected execution patterns, and program performance evaluations.

The contributions through the current studies are accomplished over multiple phases, including identification of the features used for recognizing the possible vulnerable statements in the software program. The statements are provided with an initial vulnerability rank concerning the global rank of the statements in the program. The ranks are automatically assigned to each statement in the program by tokenizing the statement. The ranks are further optimized through an aggregation-based neural ranking model that will assist the programmer's debugging and fault localization much easily and conveniently.

The entire paper is organized on the following grounds. Section 1 presents the introduction to the study on fault localization, Section 2 presents the brief literature about the existing models, and Section 3 presents the background of the proposed model, which covers the feature selection model. Section 4 offers the proposed model, and Section 5 presents the proposed model's statistical analysis concerning the other existing models. Section 6 provides the conclusion and the future perspective of the current study on bug localization.

2. Literature Review

There are various fault localization approaches available that are widely used in real-time fault diagnosis models. Statistic debugging models are one of the most predominantly used fault localization techniques. The bugs are identified through fault prediction snippets incorporated in the software [15] as presented in Liblit05. The model here works with the probability ρ in determining the fault in the software. The $fault(\rho)$ denotes the correct

fault prediction, and the $context(\rho)$ denotes the code of ρ implying the fault. All the cases whose score is $fault(\rho) - context(\rho) \leq 0$ are ignored. The leftover criteria are evaluated by respective significance ratings, indicating the association between predicates and software faults. Higher-scoring predicates must be evaluated immediately [1].

The other conventional mode of bug tracking, namely the slice-based technique [16], is performed by normalizing the software code into multiple components named the segments. Each such segment is verified for the bug by evaluating the response of the snippet. One primary use of static slicing is to assist developers in finding flaws in their systems by reducing the searching space. Because a failure could be traced to something like a variable's value being wrong, debugging can only focus on searching on the slice containing the vulnerability rather than the complete software. Using static slicing does have the drawback of including all operational snippets that may potentially influence the parameter's value in the snippet. As a consequence, it may produce an incorrect prediction. Moreover, dynamic slicing can determine the snippet that impacts a specific value seen at a particular location rather than potentially influencing such a value.

Probabilistic and empirical causation theories drive spectrum-based fault localization models [17]. If the software fails, this log data may be employed to determine the vulnerable segment of the program. It shows which snippets of the software program under test have been examined during an operation. Program state-driven bug localization is the other most predominantly used technique that keeps track of the values and the outcomes of the snippets in the software, and periodically the values are examined for the fault localization. The faults are recognized by matching the states of the development version of program snippets with a reference version. It also changes certain parameter values to check which triggers incorrect software execution.

The techniques like MULTRIC [18], TraPT [19], FLUCCS [20], and PRINCE [21] have demonstrated that Learning-to-Rank methods can assist in the identification of fault statements by using a variety of fault-diagnostic characteristics of varying dimensions. Limited in its ability to automatically choose strong preexisting features and find new advanced features for fault localization, it may not fully use the training data information gathered. The observation over the model DeepFL, which ranks suspicious logics in the program using Multi-Layer Perceptron [22]. As a result, researchers have lately begun highlighting the best of several conventional fault localization methods using machine learning to achieve more efficiency. Moreover, sophisticated deep learning techniques are used to explore powerful features for fault localization and accurately rank suspicious program snippets.

3. Background

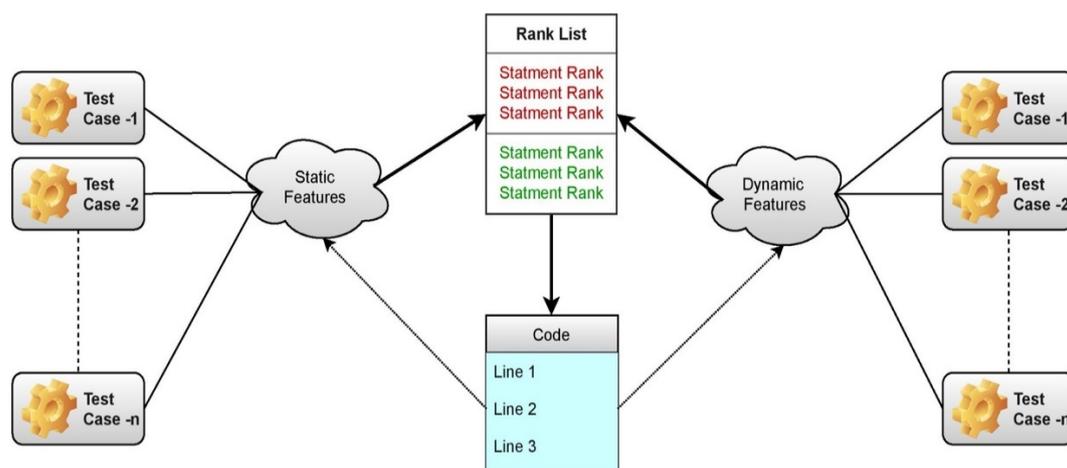
The instances in the project's implementation phase where unexpected behavior happens are failures, errors, and faults. A fault in the computer program is defined as any inappropriate move, procedure, or data specification, sometimes termed the bug. The developer may unintentionally land the bug into the program while writing. An error is a discrepancy between a calculated value and the actual value in a certain context. Failure is defined as a system failing to execute its task following the expectations of the developer. So, when the software fails to operate as expected, there must be a possible bug in the code, and if such bugs or faults are ignored, the complete software may fail.

The proposed model relies on extracting the features associated with software evaluation and assists in analyzing the working procedure of the software snippet. Thereby the features are ranked based on the probabilistic measure for the failure of the snippet. Later, the ranks are used to analyze the software's design principle and work procedure closely. Both static and dynamic features are considered for performing the fault localization and the other comprehensive feature set. The use of various classes, packages, and objects are common throughout the programming languages. Objects are created at run-time, whereas other elements are static. Their association among them is mostly dynamic [23]. A few such relations are presented in Table 1.

Table 1. Table representing the categories of various parameters.

Parameter	Static Parameter	Dynamic Parameter
Access(method, attribute)		✓
Call(method, method)		✓
Consist(class, attribute)	✓	
Consist(class, method)	✓	
Consist(class, sub-class)	✓	
Consist(package, class)	✓	
Instance(object, class)		✓
Knows(object, object)		✓
Refers (method, method)	✓	
Refers(method, Attribute)	✓	
Sub-class(class, class)	✓	
Stack Trace		✓
Dynamic Program Splitting		✓

Static methods create implementation-based requirements, whereas dynamic ones generate their specifications by watching the program run. Execution-trace analyses include dynamic strategies. The ranking model's construct is presented in Figure 1, where static and dynamic features are considered.

**Figure 1.** Represents the static and dynamic features in statement rank generation.

3.1. Static Feature Extraction

The static feature set involves the set of parameters in the software program that have a potential vulnerability that may lead to the bug in the code snippet or software failure. The feature set is all about the metadata about the program, which elucidates the complete structure of the program like the integer usage, conditional statements, the indentation of the expressions and various labels, and the annotations that are used across the code. Table 2 presents the complete set of static features considered in the bug localization process.

Table 2. Table representing the static features associated with the software.

Feature	Description of the Feature
#n_Lines	Total number of lines of code in the complete software program
#s_Lines	Total number of lines of code corresponding to snippet
#indt	Determines the level of indentation
#n_comments	Number of comments in the complete program

Table 2. Cont.

Feature	Description of the Feature
#s_comments	Number of comments in the corresponding code snippet
#n_anno	Number of annotations in the complete program
#s_anno	Number of annotations in the corresponding code snippet
#n_label	Number of labels in the complete program
#s_label	Number of labels in the corresponding code snippet
#n_g_var	Number of Global variables in the complete program
#n_l_var	Number of Local variables in the complete program
#s_g_var	Number of Global variables in the corresponding code snippet
#s_l_var	Number of Local variables in the corresponding code snippet
#n_opr	Number of operators in the complete program
#s_opr	Number of operators in the corresponding code snippet
#n_kyw	Number of keywords in the complete program
#s_kyw	Number of keywords in the corresponding code snippet
#n_null	Number of null values in the complete program
#s_null	Number of null values in the corresponding code snippet
#n_tok	Number of tokens in the complete program
#s_tok	Number of tokens in the corresponding code snippet

3.2. Dynamic Feature Extraction

The dynamic features of bug identification include the execution procedures and the outcomes associated with them. There are various bug localization mechanisms like Spectrum-Based Fault Localization [24], Predicate-based Fault Localization [25], and Program snippet analysis. The dynamic features are recognized by executing the test cases over the program, analyzing the program response to the test cases, and analyzing the stack data to trace the program's behavior. The dynamic features and the vulnerable code snippets and statements are identified through either of the models, and all such statements are ranked through an Aggregation-Based Neural ranking mechanism.

3.2.1. Spectrum-Based Fault Localization

In Spectrum-Based Fault Localization model, the possible fault and potential vulnerability ranking are assigned to each trace of program components like expressions, code statements, conditional branching statements, declarations, and assignment statements collected for each test case using a mathematical formula specific to the method. The suspiciousness rank indicates the likelihood that the statement or a snippet in the software program is defective. Using a spectrum-based fault localization strategy, every snippet's dependency information is analyzed while running test cases. The correlation and dependency information are combined with a predetermined vulnerability analysis scheme to approximate suspicious ranks for each such program element.

The ranking is based on the number of times the test cases successfully pass and failover the number of times being executed [26]. Let us assume a program P which is considered for the assessment in the current study with a set of elements which are identified as $\{p_1, p_2, \dots, p_n\}$ such that $\sum_{p=1}^n p \in P$. The formula for rank assessment is presented in Equation (1):

$$stat_{rank}(SBFL) = \frac{fail_{tc}(stat)}{\sqrt{tot_{fail_{tc}}(fail_{tc}(stat) + pass_{tc}(stat))}} \quad (1)$$

From Equation (1),

$stat_{rank}$ —Represents the statement rank

$fail_{tc}(stat)$ —Represents the fail test cases associated with the statement;

$pass_{tc}(stat)$ —Represents the passed test cases associated with the statement;

$tot_{fail_{tc}}$ —Represents the total failed test cases throughout the program.

This equation demonstrates this same fundamental concept of spectrum-based fault localization approaches: the greater the number of times failed test cases execute a statement, the greater the suspicious score of that statement; the greater the number of times a statement is executed bypassed test cases, the relatively small the suspicious score of that element; and thus illustrates the correlation among test cases and program components. The evaluated ranking model will assist in prioritizing the statement with the code snippet and assist in appropriate fault localization through the trace.

3.2.2. Predicate-Based Fault Localization

The predicate-based fault localization-based fault localization approach assesses the statements ranks in the program with a set of predetermined predicates. The program is then implemented for execution, and the values of predicates gathered throughout executions are being used to rank the statements. By combing several executions, the predicate-based approach pinpoints the key predicates associated with the preliminaries of the software failure. Equation (2) determines the predicate-based ranking for the statement within the program, and the rank was determined based on the execution of the test cases.

$$stat_{rank}(PBFL) = \frac{2}{\frac{1}{\alpha(stat)} + \frac{1}{\beta(stat)}} \quad (2)$$

where α and β are increase and sensitivity and are determined in the Equations (3) and (4):

$$\alpha = \frac{fail_{tc}(stat)}{pass_{tc}(stat) + fail_{tc}(stat)} - \frac{fail'_{tc}(stat)}{pass'_{tc}(stat) + fail'_{tc}(stat)} \quad (3)$$

$$\beta = \frac{\log(fail_{tc}(stat))}{\log(tot_{fail_{tc}})} \quad (4)$$

From Equations (3) and (4),

$fail_{tc}(stat)$ —Represents the fail test cases associated with the statement, where $stat$ has covered;

$pass'_{tc}(stat)$ —Represents the passed test cases associated with the statement, where $stat$ has covered;

The rank is determined through the harmonic mean of two factors, i.e., increase identified by $\alpha(stat)$, and sensitivity identified by $\beta(stat)$, where $\alpha(stat)$ and $\beta(stat)$ are the levels of predicate $stat$ and differentiate the feature distributions of predicate $stat$ for unsuccessful executions across all implementations that are cover over $stat$.

3.2.3. Program Snippet Analysis

The program snippet analysis is performed over the behavioral aspect of the program on executing a test case for analyzing the response in the run-time and stack trace. If the target parameter is inaccurate, it may affect some other parameters in the software. In the same way, statements with parameters or the literals in the same snippet have the potential to be fault-inducing expressions as well. Each statement's vulnerability rank is directly proportional to how often it normalizes incorrect statements. Snippet analysis may be used if there is just one failed test case. The statement that fails is used as the starting point, and then the rest of the statements are stripped off from it.

Throughout implementation, if a statement is erroneous, the program will raise an exception. The collection of active stack sessions throughout a problematic program's execution is also one of the most helpful debugging strategies [27,28]. The code snippet and associated test cases used in the validation are being presented in the Table 3 for better comprehensibility of the current study.

Table 3. Table representing the code snippet with the outcome of test cases.

sample heap.py* ×	TC-1	TC-2	TC-3	TC-4	TC-5
1 import heapq as hp	✓	✓	✓	✓	✓
2 p=[4,53,23,5,35,63]	✓	✗	✓	✓	✓
3 q=list()	✓	✓	✓	✓	✓
4 hp.heapify(p)	✓	✓	✗	✗	✓
5 # start of while loop	!	!	!	!	!
6 while True:	✓	✓	✓	✓	✓
7 q.append(heapq.heappop(p))	✓	✓	✓	✓	✗
8 if len(p)==0:	✓	✓	✓	✗	✗
9 break	✓	✓	✓	✓	✓
10 print('the sorted list:',q)	✓	✓	✓	✓	✓

The tick mark represents the pass of the test case associated with the corresponding line of the code snippet, and the cross symbol represents the test case's failure. Based on the failed test cases, the corresponding statement is given the vulnerability ranking.

3.3. Feature Set Scaling and Initial Ranking

A feature scaling approach is used to standardize the overall range of features set in input data. The feature set in the input program includes varying values throughout the learning phase while minimizing the loss function. Scaling is performed over iterations to make the localization algorithm reach the global or local best fast and precisely. In the current study, the min-max normalization is performed in scaling the feature values in the range 0–1 [29]. The normalizing approach known as Min-Max brings numerous advantages over traditional scaling techniques. Min-Max scaling is capable of handling the non-Gaussian feature distribution. Min-Max normalization is made to solve the loss of precision in a method that optimizes the gradient while moving toward the global solution. It produces target values ranging from 0 to 1 by taking the column's min and max values, as illustrated in Equation (5):

$$f_{new} = \frac{f - f_{min}}{f_{max} - f_{min}} \quad (5)$$

From Equation (5), the variable f_{new} denotes the new normalized values in the range 0–1, f_{min} denotes the smallest values associated with the corresponding feature and the variable f_{max} denotes the largest value associated with the particular feature. The variable f denotes the corresponding data sample. Now for fine-tuning the ranks of each of the features that are identified, the ranking is performed in concern to the global best, resultantly the vulnerability rank of the current feature is following the other features across the complete program, and it does the job of localizing the bugs based on the impact of the bug on the software [30,31].

$$stat_{Rank} = stat_{Rank-1} + G_Best_stat_{Rank} - L_Best_stat_{Rank-1} \quad (6)$$

From Equation (6), the variable $stat_{Rank}$ denotes the vulnerability rank of the statement at the corresponding iteration and the variable and $stat_{Rank-1}$ denote the rank associated with the corresponding statement in the previous iteration. The variable $G_Best_stat_{Rank}$ denotes the global best vulnerability ranking across the complete program and similarly the variable $L_Best_stat_{Rank-1}$ denotes the local best vulnerability rank within the code snippet associated with the software program. The ranks on updating concerning the global best rank, the more vulnerable statements through the program are prioritized to resolve first before the less significant statements.

4. Proposed Model

The proposed ABNR model works are mechanized to identify the features based on the significance using the self-attention layer through the weighted approach, and aggregation is performed in the neural ranking process. The statements are then ranked based on the

severity of the vulnerability using the softmax layer of the neural ranking model [32]. The block diagram of the proposed model is presented in Figure 2 [33]. The code snippets that are the part of the software program are fed as the input for the neural ranking model. The neural ranking model identifies the features from the data and assigns the weights to the features based on the importance of those feature in the fault determination. The features are then correlated with the features that are already trained to the model using the feature map and the probabilities are assessed. Then based on the probabilities the ranks are provided to the statements.

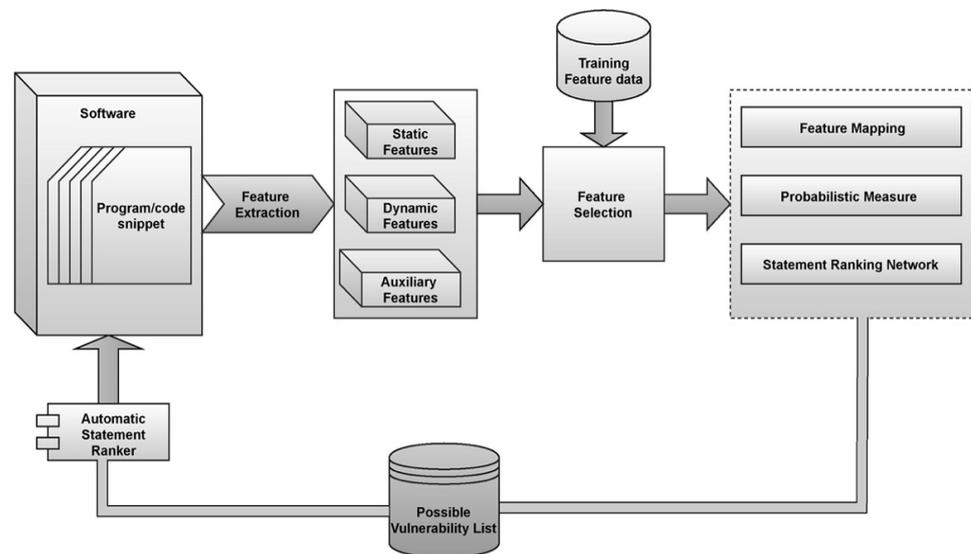


Figure 2. Represents the block diagram of the proposed fault localization model.

4.1. Layered Architecture of the Aggregation-Based Neural Ranking

The layered model of the ABNR neural ranking model consists of multiple layers that would determine the rank of the statement associated with the vulnerability. The model architecture includes the self-attention layer, neural ranking layer through aggregation, and the SoftMax layer approximating the suitable rank for the statement. The model is trained with the data with similar statements with relevant vulnerabilities.

Neural Ranking techniques have subsequently been presented for determining the relevance of a susceptible statement to a code snippet by examining the vulnerability statements, patterns of statement fragments matching in the training data, or a combination of the two. By seeing a huge variety of vulnerable and normal code samples during training, these models often learn to discriminate between the code feature distributions associated with a pertinent and a less pertinent vulnerable statement-code snippet combination. Compilations of statements in each category provide a partial order for the statements in that list. Assigning a numeric score for every such statement usually induces the kind of vulnerability ranking. The neural ranking technique aims to rank program statements in an unseen lists manner comparable to how scores were determined from the training samples.

Self-Attention Layer

The self-attention layer is mechanized to integrate several feature sources and allot suitable weights to each source of information for fault localization. The neural ranking model assigns the erroneous statement the maximum rank. Attributed to the reason that various features have varying degrees of significance, the self-attention layer is utilized to integrate and improve important data from static and dynamic features. The normalization layer separates all three kinds of features, semantic, statistical, and dynamic. The self-attention layer combines all three vectors and assigns the vector to the $f(x)$, $g(x)$, and $h(x)$ components [33]. Each layer would subsequently distinguish inputted features and

produce a feature map. Self-attention connects data from various places within input pattern to compute the scaled dot product attention, as shown in Equation (7):

$$attention(f(x), g(x), h(x)) = softmax \left[\frac{(f(x) \times g(x))^T}{\sqrt{d_{g(x)}}} \right] \times h(x) \quad (7)$$

$$f(x) = \mathbb{R}^{f_i \times f_d} \quad (8)$$

$$g(x) = \mathbb{R}^{g_i \times g_d} \quad (9)$$

$$h(x) = \mathbb{R}^{h_i \times h_d} \quad (10)$$

From the above Equations (7)–(10), the components $f(x)$, $g(x)$, $h(x)$ are the components associated with statements, test cases, and values, respectively. The variables f_i, g_i, h_i denote the i th element of each of that feature, and the variables f_d, g_d, h_d denote the dimension associated with each of those features. The variable d denotes the spatial dimension with finite set of features. The value $\sqrt{d_{g(x)}}$ divides the dot product of the snippets with the possible vulnerable statements, and softmax is applied to obtain the associated weights of each such feature. Earlier studies hypothesize that when d increases, the amplitude of the dot product increases, forcing the softmax into areas with very tiny gradients. To compensate for this impact, multiply the dot products by $\frac{1}{\sqrt{d_{g(x)}}}$ that avoid softmax converging toward less significant features. Typically, every statement's rank is calculated as a weighted sum of associated values, for each weight of the value being determined by an objective statement for the statement using a homologous test case [34]. The layered architecture of the proposed model is presented in Figure 3.

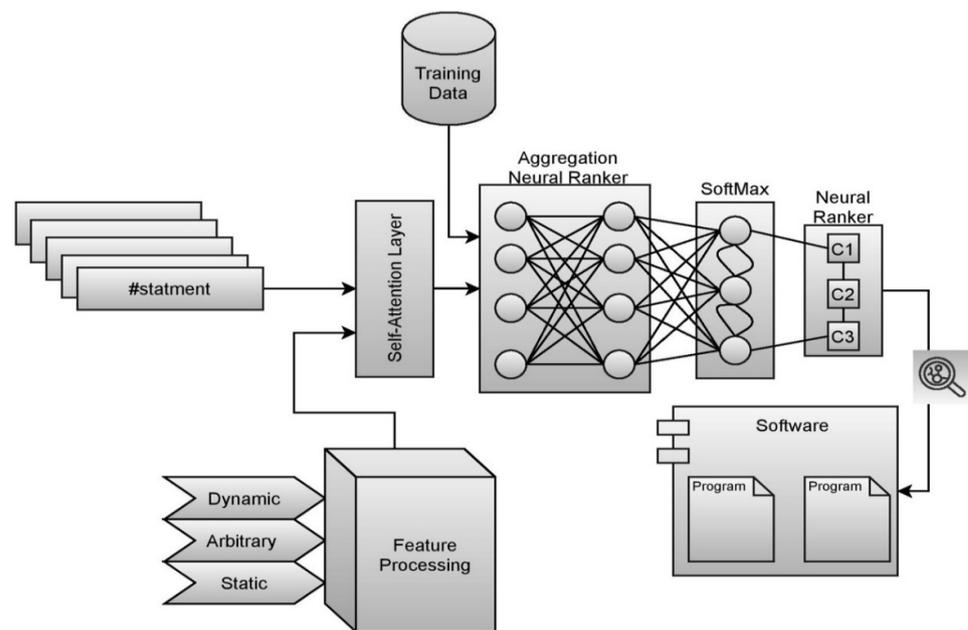


Figure 3. Represents the layered architecture of the fault localization model.

4.2. Weights Assessment Procedure

Weighing characteristics to enhance class labeling accuracy is possible when various factors affect the class label differently. The normalized mutual information (NMI) [35] between each feature and the class label as the feature's weight since mutual information measures the independence of two variables. To better understand the NMI-based weight assignment over the dataset D with T instances, which comprises two parameters p and

q with m, n instances, respectively, over the class label c [36]. The weights for both the features are determined through the following Equations (11) and (12):

$$\omega(p) = \frac{\alpha(p, c)}{\text{mean}(\beta(p), \beta(c))} \quad (11)$$

$$\omega(q) = \frac{\alpha(q, c)}{\text{mean}(\beta(q), \beta(c))} \quad (12)$$

From Equations (11) and (12), the variable α denotes the mutual information. Furthermore, the variable β denotes the entropy. The mutual information is determined through the Equations (13) and (14):

$$\alpha(p, c) = \sum_{i=0}^{m-1} \sum_{j=0}^{k-1} \frac{|p_i \cap c_j|}{T} \log \left(\frac{T|p_i \cap c_j|}{|p_i||c_j|} \right) \quad (13)$$

$$\alpha(q, c) = \sum_{i=0}^{m-1} \sum_{j=0}^{k-1} \frac{|q_i \cap c_j|}{T} \log \left(\frac{T|q_i \cap c_j|}{|q_i||c_j|} \right) \quad (14)$$

Entropy, as it pertains to machine learning, is a measure of the unpredictability and randomness of the analyzed information. The bigger the entropy, the more difficult it is to make any inferences out from the information. The entropy is being assessed through the Equations (15)–(17).

$$\beta(p) = - \sum_{i=0}^{m-1} \rho \left(\frac{|p_i|}{T} \right) \log \left(\frac{|p_i|}{T} \right) \quad (15)$$

$$\beta(q) = - \sum_{i=0}^{m-1} \rho \left(\frac{|q_i|}{T} \right) \log \left(\frac{|q_i|}{T} \right) \quad (16)$$

$$\beta(c) = - \sum_{i=0}^{m-1} \rho \left(\frac{|c_i|}{T} \right) \log \left(\frac{|c_i|}{T} \right) \quad (17)$$

4.3. Aggregation Based Neural

The aggregation-based neural ranking model for the bug localization is robust in assigning the ranks to the statements by considering the other statements within the code snippet. The principle logic in the model is that if any of the statements are erroneous, the consecutive statements after that are influenced by the erroneous statement. Resultantly the scores of the statements are updated following the previous nodes along with their vulnerability score. However, the scores of the statements are updated on normalizing each statement and correcting them where so ever required. The mathematical model for the same is presented in the current subsection of the study.

To better understand the process flow of the fault localization model, consider the following diagram, which has four statements in the snippet labeled $A, B, C,$ and $D,$ in which the erroneous origin statement is depicted by $A.$ Statement D represents the data-receiving destination. It is believed that the intermediate statements are located between the statements and the nodes intended to be reached by the data. The fault rank is used to determine the vulnerability associated with each corresponding statement. This procedure is described in detail in Equations (18)–(23).

$$R(S_A) = \{(Int_{\omega}(A) \times \rho(A)) + \Delta(\omega_{A,B} + \omega_{A,C} + \omega_{A,D})\} \quad (18)$$

$$R(S_B) = \{\rho(B) - \Delta(\omega_{A,B} + \omega_{B,C} + \omega_{B,D})\} \quad (19)$$

$$R(S_C) = \{\rho(C) - \Delta(\omega_{A,C} - \omega_{B,C} + \omega_{C,D})\} \quad (20)$$

$$R(S_D) = \{\rho(D) - \Delta(\omega_{A,D} - \omega_{B,D} - \omega_{C,D})\} \quad (21)$$

From Equations (18)–(21), various variables are being used to demonstrate the ranking procedure by aggregating the weights associated with each statement.

$R(S_A)$ —Rank associated with statement A ;

$R(S_B)$ —Rank associated with statement B ;

$R(S_C)$ —Rank associated with statement C ;

$R(S_D)$ —Rank associated with statement D ;

$Int_{\omega}(A)$ —Initial weights associated with node A ;

$\rho(A)$ —Vulnerability probability of statement A ;

$\rho(B)$ —Vulnerability probability of statement B ;

$\rho(C)$ —Vulnerability probability of statement C ;

$\rho(D)$ —Vulnerability probability of statement D ;

Δ —The overall difference among the weights associated with statements;

$\omega_{i,j}$ —Weigh the difference among the statements ' i ' and ' j '.

The aggregation-based ranking approach is reasonably fair in approximating the vulnerability score is concerning the other statements within the same code snippet. The initial probability is approximated through the softmax scores associated with the statement. The working procedure of the aggregation model is demonstrated in Figure 4.

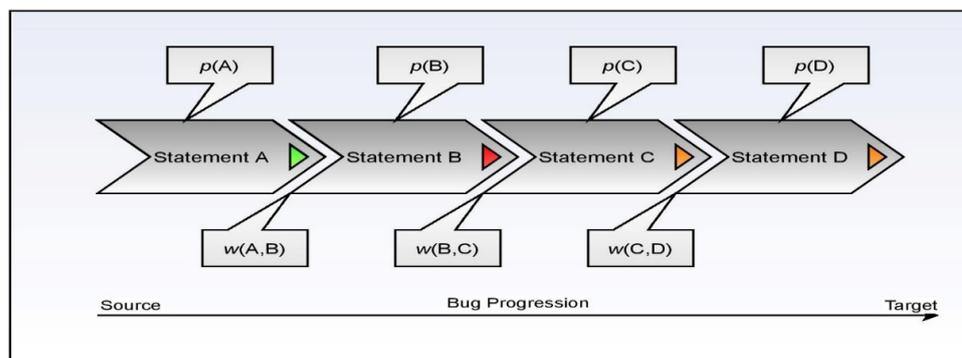


Figure 4. Represents the aggregative score model for fault localization.

The sample test case for evaluating the division of two numbers is presented through the Algorithm 1. The outcome of the test case would be a pass or fail based on the provided data. The test case would be evaluated against the data, and many such test cases are associated with each such code snippet for evaluating the vulnerability score of the statement.

Algorithm 1 (Algorithm of sample test cases)

Input: Two input variables for performing the division and auxiliary variables

Output: The return values like the quotient and remainder.

The Boolean values on executing the test case.

Step: Evaluate_Division_two_Numbers

```

x:=7.0;
y:=4.0;
Run(Div:=q);
    Assert.equal(b, remainder);
    Assert.equal(valid,true);
Else
    Assert.equal(valid,false);

```

End Step

Begin sample_test_case (Passing)

Let $\check{S}^+ \in S$ denotes the set of passes test cases in S

$\check{S} \leftarrow \text{Sample}(\check{S})^+$

Begin sample_test_case (failing)

Let $\check{S}^- \in S$ denotes the set of passes test cases in S

$\check{S} \leftarrow \text{Sample}(\check{S})^-$

Return $((\check{S})^+ \cup (\check{S})^-)$

4.4. Dataset Acquisition and Experimental Setup

Siemens test case topic applications [37] and four Unix utility programmed like gzip, sed, flex, and grep are considered to assess the proposed fault localization model [38]. The data are the part of Typically, and many researchers have used these theme programs for fault localization. Siemens test case programs are used for one fault, whereas Unix utility applications include actual and injected flaws obtained from [39]. Each topic program in the Siemens validation set contains around 1000 test inputs and comprises seven distinct test programs: tcas, tot info, schedule2, print tokens, print tokens2, replace, and tot info2. File compression and decompression are handled by the gzip program in the Unix real-life utility program. Name files may benefit from the program's ability to shrink their size, which is why it is often used. The gzip program's input consists of 13 different parameters and a list of files to compress. With 6573 lines of code and 211 test inputs, the software can do a lot. The sed software is used to make small modifications to a stream of input. It is mostly used for parsing text input and making user-specified modifications to that data. There are 360 test inputs and 12,062 lines of code in the program.

A lexical analyzer is what the flex program does. The input files were created of rules, which are collections of regular expressions and C code. There are 13,892 lines of code in all, and 525 test inputs are provided. Patterns and files are the two input parameters for the grep program. Every file that includes a match to one of the patterns has lines printed by the software. Lines of code: There are 12,653, and 470 inputs provided [40]. Real and injected errors will now be included in Unix utility applications. The details of the subject programs considered in the proposed model evaluation are presented in Table 4 [41,42]. The normal and vulnerable program statements are the samples that are used in the both the training and the validation phase of the proposed model. Training data contains 60% of the original data samples of the code. The overall samples are indeed made up of a 60% training dataset and a 40% validation dataset chosen at random from normal and defective samples. The remaining 40% of the original dataset is the testing dataset assumed as the unseen portions used to assess the ABNR fault localization performance.

Table 4. Table representing the subject programs for evaluation.

Program	Number of Vulnerable Snippets	Number of Lines in the Code	Role	Test Cases
print_token	7	565	Lexical analyzer	4130
replace	32	412	Sequence replacement	2650
schedule	10	307	Priority scheduler	2710
tcas	41	173	Altitude separation	1608
Sed	7	12,062	Text manipulator	360
tot_info	23	406	Information Measure	1052
Gzip	5	6573	Data compression	211
Flex	22	13,892	Lexical analyzer	525
Grep	7	12,653	Sequence searcher	470

The performance of the proposed ABNR model is evaluated by executing multiple test cases over the code snippets. The test cases assess the validity of the code snippet under variable factors including the divergent inputs and operational conditions. The evaluations are carried locally in the standalone computer by deploying the necessary software. The details of the experimental environment where the experimentation is carried is presented in Table 5.

Table 5. Information about the implementation environment.

Environment Details	Specification
Operating System	Microsoft Windows 10
Processor	Intel Xeon E5-2687W
Architecture	64-bit
Memory allotted	720MB
GPU	NVIDIA Quadro P1000
Cuda-Parallel Processing cores	640
Language	Python
Libraries used	Pandas, Numpy, Scikit

5. Results and Discussion

To assess a defect localization approach's efficiency, it is critical to use appropriate measurements. The aggregate number of assertions is evaluated, Wilcoxon signed-rank test and Top-N are all used in the current study. If it is to be determined, an appropriate measure should be employed to assess the relevance and usefulness of a defect localization method. The score is a criterion that is defined as the proportion of code snippet that does not need to be inspected to identify a flaw in the program. The exam score is defined as the percentage of code that has to be inspected until the first statement in which the problem is located is reached during the examination process. The exam score (ES) is often used in many research to assess the efficacy of a fault localization method [43,44]. A method α with a lower exam score than that of another approach β will be regarded as being more successful in comparison to technique B since technique α requires fewer code statements to be inspected to identify any flaws than technique β . The ES values are mathematically evaluated using the Equation (22), concerning the vulnerability score of the statement recognized by V_s_Stat .

$$ES = \frac{V_s_Stat}{Num_of_lines_of_code} \quad (22)$$

The cumulative number of statements that must be evaluated concerning subject programs to identify errors is taken into account [45]. So, for any given programs with n faulty versions, $A(k)$ and $B(k)$ are the percentages of statements that must be reviewed for two fault localization techniques, A and B, to detect all defects inside the kth Faulty version of the program. Approach B is more effective than technique A when it requires a programmer to analyze fewer statements to find all flaws in the erroneous versions, as shown in Equation (23); procedure B seems to be efficient over procedure A.

$$\sum_{k=1}^n B(k) < \sum_{k=1}^n A(k) \quad (23)$$

In addition, the Wilcoxon signed-rank test [46] is used to provide a rigorous empirical assessment of methodologies in terms of their efficacy. Top-N represents the proportion of errors within Top N (N = 1, 5, 10) places that a fault localization method ranks for any problematic code snippet. As a result, the measure becomes stronger as the magnitude of N in Top-N decreases [47]. The performance of the proposed model is compared against the similar fault localization models Jaccard [48], Ochiai [48,49], Tarantula [48,50], software-network centrality measure (SNCM) [42], fault localization technique based on complex network theory (FLCN-S) [40]. The total amount of statements evaluated in Siemens' validation set and Unix utility applications to find errors are presented in Table 6.

Table 6. Denote the number of statements analyzed for fault detection from Siemen’s validation set.

	print_token	replace	schedule	schedule 2	Tot_info	Tcas
Jaccard	602	457	569	580	456	380
Ochiai	518	435	533	538	345	307
Tarantula	570	492	525	571	425	363
SNCM	658	567	612	651	600	660
FLCN-S	475	398	488	504	275	292
ABNR	431	322	407	468	198	253

The proposed ABNR model outperforms compared to the other existing models for fault localization. The performance is determined based on the number of statements being evaluated to determine the faulty code snippet. The model capable of recognizing the vulnerability with fewer statements is assumed to be efficient with lesser computational efforts and faster response. From Table 6, it can be observed that the proposed model is comparatively better than the other model considered. The graphs are generated from tabular data in Figure 5.

**Figure 5.** Graphs representing the number lines of code evaluated for fault localization for each category in Siemen’s validation set.

The performance of the proposed ABNR model is also evaluated against the Unix utility application programs. The model has recognized the faulty statements in the code snippet by evaluating the fewer lines of code, which needs comparatively lesser

computational efforts and is much faster than its counterparts. The details about the number of lines of code evaluated for Unix applications are presented in Table 7, and the corresponding graphs are presented in Figure 6.

Table 7. Denotes the number of statements analyzed for fault detection in Unix utility applications.

	gzip	sed	flex	grep
Ochiai	3342	4269	1497	3959
FLCN-S	2357	3651	1298	1848
ABNR	2109	3189	997	1241



Figure 6. Graphs representing the number lines of code evaluated for fault localization for each category in the Unix utility program.

It can be observed from the values presented in Tables 6 and 7, that the proposed aggregation-based neural ranking model performs as desired by assigning the ideal ranks to the vulnerable statements that would assist in fault localization by verifying the fewer statements. The proposed model is further evaluated through the Wilcoxon signed-rank test concerning the confidence value of similarity among two independent samples of code to compare overall sample average rankings of two samples using paired differential tests. The ABNR has been evaluated like the other models such as Jaccard, Ochiai, Tarantula. The comparative analysis of the confidence measures is presented in Table 8, and Figure 7 represents the graphs generated from the evaluated confidence values. The proposed ABNR model has exhibited better confidence in evaluating the fault statements in the code snippet.

Table 8. Denote the confidence level of fault detection from Siemen’s validation set.

	print_token	replace	schedule	schedule 2	Tot_info	Tcas
Jaccard	99.17	98.32	98.83	98.33	99.33	98.77
Ochiai	95.47	97.22	97.88	97.88	98.58	92.5
Tarantula	98.66	98.45	97.63	98.36	99.24	98.43
ABNR	99.43	99.12	99.65	99.21	99.57	99.23

**Figure 7.** Graphs representing the confidence level of fault localized for each category in Siemen’s validation set.

The Wilcoxon signed-rank test is performed over the Unix application programs to assess the confidence of similarity among two independent code samples. The assessed

values for the same are presented in Table 9, and the generated graphs are presented in Figure 8.

Table 9. Denote the confidence level of fault detection from Unix utility applications.

	gzip	sed	flex	Grep
Ochiai	99.89	99.84	99.49	99.95
FLCN-S	99.92	99.89	99.56	99.97
ABNR	99.95	99.91	99.86	99.97



Figure 8. Graphs representing the confidence level of fault localized for each category in Unix utility applications.

The efficiency of the Fault localization approaches is evaluated based on their ability to locate actual flaws. They also perform significantly better at incorporating the right response over top N statements of their outcome. Usually, N would be 5 or 10. The study has found that concentrating on factors other than location would lead to substantial advancements in fault localization. The assessed values concerning Top-N statements are presented in Table 10.

Table 10. Denote the Top-N assessment for faults localized in Unix utility applications.

	Top-N	gzip	sed	flex
Ochiai	5	29.15	32.82	29.11
	10	44.27	40.67	39.12
FLCN-S	5	35.62	41.01	48.01
	10	51.01	49.00	58.44
ABNR	5	41.21	48.27	54.11
	10	57.89	54.20	66.02

It can be observed from the above table that the performance of the ABNR is better compared to that of the counterparts as the proposed model is capable of localizing the vulnerable statements more appropriately compared to the other approaches. The

evaluations on the proposed model have made it clear that the model is more precise with lesser computational efforts in fault localization.

5.1. Practical Implication

The proposed fault localization model through the aggregation-based neural ranking method is incorporated in the code editor names code debugger to evaluate the faults in the code snippet. The fault localization model would update the ranks for each corresponding statement in the code snippet on evaluating the code. The color box represents the severity of the error or bug corresponding to the code. The orange color denotes the possible fault or the vulnerability associated with the code statement, and the red color represents the faulty statement. The green color represents the error-free statements, and the variable “snp” denotes the snippet of the corresponding program. Figure 9 denotes the front end of the practical implication model.

Code Snippet	Severity	Score	Memory	Snippet Count
1.datagen = ImageDataGenerator(Orange	R-0.621	5.6B	snp : 2
2.rotation_range=180,	Green	R-0.071	2B	snp : 2
3.height_shift_range=0.1,	Green	R-0.096	3.2B	snp : 2
4.zoom_range=0.1,	Green	R-0.178	3.2B	snp : 2
5.horizontal_flip=True,	Green	R-0.164	1.2B	snp : 2
6.fill_mode='nearest')	Orange	R-0.621	2.7B	snp : 2
7.img_path = load_img('../input/ABC/D.jpg')	Green	R-0.621	14B	snp : 2
8.x = img_to_array(img_path)	Orange	R-0.525	6.9B	snp : 2
9.x = x.reshape((1,) + x.shape)	Orange	R-0.581	1.8B	snp : 2
!10.layer_names = [layer.name for layer in model.layers]	Red	R-0.893	7.4B	snp : 2
11.dst = os.path.join(img_dir, fname)	Green	R-0.463	23B	snp : 2
12.y = np.clip(x, 0, 255).astype('uint8')	Green	R-0.459	106B	snp : 2

Figure 9. Image of the code editor with fault localization model.

The dashboard embedded in the dashboard would be immediate assistance for the developer to assess the statement’s vulnerability. The summary module is associated with the editor that would conclude the metadata associated with the code snippet executed. The developer can obtain the summarized information about the code snippet at a single point. Figure 10 that is presented below denotes the summarized information about the code snippet.

Code Snippet	Severity	Score	Memory	Snippet Count
1.datagen = Im	Orange	R-0.621	5.6B	snp : 2
2.rotation_far	Green	R-0.071	2B	snp : 2
3.height_shift	Green	R-0.096	3.2B	snp : 2
4.zoom_range=0	Green	R-0.178	3.2B	snp : 2
5.horizontal_f	Green	R-0.164	1.2B	snp : 2
6.fill_mode='r	Orange	R-0.621	2.7B	snp : 2
7.img_path = l	Green	R-0.621	14B	snp : 2
8.x = img_to_e	Orange	R-0.525	6.9B	snp : 2
9.x = x.reshap	Orange	R-0.581	1.8B	snp : 2
!10.layer_names	Red	R-0.893	7.4B	snp : 2
11.dst = os.pa	Green	R-0.463	23B	snp : 2
12.y = np.clip	Green	R-0.459	106B	snp : 2

Figure 10. Image summarized report about the code snippet.

The proposed technology for fault localization integrated into the editor would assist the developer in faster fault localization. Upon successfully coding a function in the program or the code snippet of an application, the generated summarized report would assist the developer in ease of debugging the program.

5.2. Threats to Validity

The proposed ABNR model is confined to a single error approximation and assigns the ranks to the statements; all the statements that come after the vulnerable statements are influenced by the erroneous statements. But the ABNR is limited for assessing the ranks from the single statement error that can be further improvised by multi-error prediction mechanism. The multi-error prediction model would assist in precise rank of the statements for better localization probability.

6. Conclusions

The current study proposes a novel and efficient bug localization approach through an aggregation-based neural ranking technique. The proposed model considers the vulnerable statement, and ranking is provided to all the consecutive statements based on the faulty statement, which will assist the developers in quickly and precisely pointing out the vulnerability. The proposed approach on evaluating the benchmark subject programs like the Siemens validation set and Unix Utility applications shows reasonable performance than the counterparts. The model is evaluated across the divergent metric like the assertions evaluated, Wilcoxon signed-rank test, and Top-N. The results have proven that the ABNR model can localize the faults by evaluating fewer statements with better confidence. The weights optimization would yield better performance, and incorporating the auxiliary memory components for maintaining the state information would enhance the performance of the fault localization model. The practical implication model shown in the results and discussion section is expected to be the common feature for the majority of the code editors in the future generation. Yet, there is good scope for research in assessing the overall rank of the code snippet based on the faulty statements, which would determine the overall quality of the software.

Funding: The author has not received any specific funding for this study. This pursuit is a part of his scholarly endeavors.

Data Availability Statement: The data for evaluation of the model is being acquired from an open-source repository, namely Software-artifact Infrastructure Repository (SIR), that consist of the subject programs like print token, schedule, schedule 2, replace, tcas, tot_info that are part of Siemen's validation set and programs like gzip, sed, flex, and grep that are the part of the Unix utility suite. The programs are associated with the test cases that would assist in tracing the vulnerable statements.

Acknowledgments: The author would like to acknowledge the College of Computer Sciences and Information Technology, King Faisal University, Saudi Arabia, for providing the infrastructure to carry out the research.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 707–740. [[CrossRef](#)]
2. Xiao, X.; Pan, Y.; Zhang, B.; Hu, G.; Li, Q.; Lu, R. ALBFL: A novel neural ranking model for software fault localization via combining static and dynamic features. *Inf. Softw. Technol.* **2021**, *139*, 106653. [[CrossRef](#)]
3. Zhang, W.; Li, Z.; Wang, Q.; Li, J. FineLocator: A novel approach to method-level fine-grained bug localization by query expansion. *Inf. Softw. Technol.* **2019**, *110*, 121–135. [[CrossRef](#)]
4. Zou, D.; Liang, J.; Xiong, Y.; Ernst, M.D.; Zhang, L. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Trans. Softw. Eng.* **2021**, *47*, 332–347. [[CrossRef](#)]
5. Ferenc, R.; Bán, D.; Grósz, T.; Gyimóthy, T. Deep learning in static, metric-based bug prediction. *Array* **2020**, *6*, 100021. [[CrossRef](#)]
6. Zhang, W.; Wang, S.; Wang, Q. BAHA: A Novel Approach to Automatic Bug Report Assignment with Topic Modeling and Heterogeneous Network Analysis. *Chinese J. Electron.* **2016**, *25*, 1011–1018. [[CrossRef](#)]

7. Jeong, G.; Kim, S.; Zimmermann, T. Improving bug triage with bug tossing graphs. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09). Association for Computing Machinery, New York, NY, USA, 24–28 August 2009; pp. 111–120. [[CrossRef](#)]
8. Ericwong, W.; Qu, Y. Bp Neural Network-Based Effective Fault Localization. *Int. J. Softw. Eng. Knowl. Eng.* **2011**, *19*, 573–597. [[CrossRef](#)]
9. Ghosh, D.; Singh, J. Spectrum-based multi-fault localization using Chaotic Genetic Algorithm. *Inf. Softw. Technol.* **2021**, *133*, 106512. [[CrossRef](#)]
10. Maru, A.; Dutta, A.; Kumar, K.V.; Mohapatra, D.P. Software fault localization using BP neural network based on function and branch coverage. *Evol. Intell.* **2021**, *14*, 87–104. [[CrossRef](#)]
11. Rothenberg, B.C.; Grumberg, O. Must Fault Localization for Program Repair. In *Computer Aided Verification. CAV 2020. Lecture Notes in Computer Science*; Lahiri, S., Wang, C., Eds.; Springer: Cham, Switzerland, 2020; Volume 12225. [[CrossRef](#)]
12. Lin, C.-T.; Chen, W.-Y.; Intasara, J. A Framework for Improving Fault Localization Effectiveness Based on Fuzzy Expert System. *IEEE Access* **2021**, *9*, 82577–82596. [[CrossRef](#)]
13. Choi, K.-Y.; Lee, J.-W. Fault Localization by Comparing Memory Updates between Unit and Integration Testing of Automotive Software in an Hardware-in-the-Loop Environment. *Appl. Sci.* **2018**, *8*, 2260. [[CrossRef](#)]
14. Zhou, C.; Kumar, R.; Jiang, S. Analysis of run-time data-log for software fault localization. In Proceedings of the 2011 American Control Conference, San Francisco, CA, USA, 29 June–1 July 2011; pp. 5127–5132. [[CrossRef](#)]
15. Liblit, B.; Naik, M.; Zheng, A.X.; Aiken, A.; Jordan, M.I. Scalable statistical bug isolation. In Proceedings of the 2005 ACM SIGPLAN Conference On Programming Language Design and Implementation (PLDI '05). Association for Computing Machinery, New York, NY, USA, 12–15 June 2005; pp. 15–26. [[CrossRef](#)]
16. Xu, B.; Qian, J.; Zhang, X.; Wu, Z.; Chen, L. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes* **2005**, *30*, 1–36. [[CrossRef](#)]
17. Troya, J.; Segura, S.; Parejo, J.A.; Ruiz-Cortés, A. Spectrum-Based Fault Localization in Model Transformations. *ACM Trans. Softw. Eng. Methodol.* **2018**, *27*, 1–50. [[CrossRef](#)]
18. Xuan, J.; Monperrus, M. Learning to Combine Multiple Ranking Metrics for Fault Localization. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 28 September–3 October 2014; pp. 191–200. [[CrossRef](#)]
19. Li, X.; Zhang, L. Transforming programs and tests in tandem for fault localization. In Proceedings of the ACM on Programming Languages, New York, NY, USA, 12 October 2017; Volume 1, pp. 1–30. [[CrossRef](#)]
20. Sohn, J.; Yoo, S. FLUCCS: Using code and change metrics to improve fault localization. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017; pp. 273–283. [[CrossRef](#)]
21. Kim, Y.; Mun, S.; Yoo, S.; Kim, M. Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs. *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*, 1–34. [[CrossRef](#)]
22. Li, X.; Li, W.; Zhang, Y.; Zhang, L. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 169–180. [[CrossRef](#)]
23. Löwe, W.; Ludwig, A.; Schwind, A. Understanding software-static and dynamic aspects. In Proceedings of the 17th International Conference on Advanced Science and Technology, Mielno, Poland, 17–19 October 2001; pp. 17–18.
24. Sasaki, Y.; Higo, Y.; Matsumoto, S.; Kusumoto, S. SBFL-Suitability: A Software Characteristic for Fault Localization. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, 27 September–3 October 2020; pp. 702–706. [[CrossRef](#)]
25. Jiang, J.; Wang, R.; Xiong, Y.; Chen, X.; Zhang, L. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 10–15 November 2019; pp. 502–514. [[CrossRef](#)]
26. Abreu, R.; Zoetewij, P.; Golsteijn, R.; van Gemund, A.J.C. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* **2009**, *82*, 1780–1792. [[CrossRef](#)]
27. Wong, C.; Xiong, Y.; Zhang, H.; Hao, D.; Zhang, L.; Mei, H. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 28 September–2 October 2014; pp. 181–190. [[CrossRef](#)]
28. Jiang, S.; Li, W.; Li, H.; Zhang, Y.; Zhang, H.; Liu, Y. Fault Localization for Null Pointer Exception Based on Stack Trace and Program Slicing. In Proceedings of the 2012 12th International Conference on Quality Software, Washington, DC, USA, 27–29 August 2012; pp. 9–12. [[CrossRef](#)]
29. Alzahrani, A.O.; Alenazi, M.J.F. Designing a Network Intrusion Detection System Based on Machine Learning for Software Defined Networks. *Future Internet* **2021**, *13*, 111. [[CrossRef](#)]
30. Naga Srinivasu, P.; Balas, V.E. Self-Learning Network-based segmentation for real-time brain M.R. images through HARIS. *Peer J. Comput. Sci.* **2021**, *7*, e654. [[CrossRef](#)] [[PubMed](#)]
31. Srinivasu, P.N.; Rao, T.S.; Balas, V.E. A systematic approach for identification of tumor regions in the human brain through HARIS algorithm. In *Deep Learning Techniques for Biomedical and Health Informatics*; Academic Press: Cambridge, MA, USA, 2020. [[CrossRef](#)]

32. Deepalakshmi, P.; Lavanya, K.; Srinivasu, P.N. Plant Leaf Disease Detection Using CNN Algorithm. *Int. J. Inf. Syst. Modeling Des. (IJISMD)* **2021**, *12*, 1–21. [[CrossRef](#)]
33. Chen, Z.; Tong, L.; Qian, B.; Yu, J.; Xiao, C. Self-Attention-Based Conditional Variational Auto-Encoder Generative Adversarial Networks for Hyperspectral Classification. *Remote Sens.* **2021**, *13*, 3316. [[CrossRef](#)]
34. Li, L.; Lu, Z.; Watzel, T.; Kürzinger, L.; Rigoll, G. Light-Weight Self-Attention Augmented Generative Adversarial Networks for Speech Enhancement. *Electronics* **2021**, *10*, 1586. [[CrossRef](#)]
35. Vinh, L.T.; Lee, S.; Park, Y.T.; d’Auriol, B.J. A novel feature selection method based on normalized mutual information. *Appl. Intell.* **2012**, *37*, 100–120. [[CrossRef](#)]
36. Bagui, S.; Wang, T.; Bagui, S. A Novel Weighting Attribute Method for Binary Classification. *Adv. Comput.* **2021**, *11*, 1–9. [[CrossRef](#)]
37. Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T. Experiments on the effectiveness of dataflow—And control-flow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 16–21 May 1994; pp. 191–200. [[CrossRef](#)]
38. Jiang, B.; Zhang, Z.; Chan, W.; Tse, T.H.; Chen, T. How Well Does Test Case Prioritization Integrate with Statistical Fault Localization? *Inf. Softw. Technol.* **2012**, *54*, 739–758. [[CrossRef](#)]
39. Software-Artifact Infrastructure Repository. Available online: <https://sir.csc.ncsu.edu/php/previewfiles.php> (accessed on 29 October 2021).
40. Zakari, A.; Lee, S.; Hashem, I. A single fault localization technique based on failed test input. *Array* **2019**, *3–4*, 100008. [[CrossRef](#)]
41. Lanzaro, A.; Natella, R.; Winter, S.; Cotroneo, D.; Suri, N. An Empirical Study of Injected versus Actual Interface Errors. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014), Bay Area, CA, USA, 21–25 July 2014. [[CrossRef](#)]
42. Zakari, S.A.; Lee, P.; Chong, C.Y. Simultaneous Localization of Software Faults Based on Complex Network Theory. *IEEE Access* **2018**, *6*, 23990–24002. [[CrossRef](#)]
43. Singh, P.K.; Garg, S.; Kaur, M.; Bajwa, M.S.; Kumar, Y. Fault localization in software testing using soft computing approaches. In Proceedings of the 2017 4th International Conference on Signal Processing, Computing and Control (ISPCC), Solan, India, 21–23 September 2017; pp. 627–631. [[CrossRef](#)]
44. Zhu, L.; Yin, B.; Cai, K. Software Fault Localization Based on Centrality Measures. In Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops, Munich, Germany, 18–22 July 2011; pp. 37–42. [[CrossRef](#)]
45. Wang, T.; Wang, K.; Su, X. Fault localization by analyzing failure propagation with samples in cloud computing environment. *J. Cloud Comp.* **2020**, *9*, 17. [[CrossRef](#)]
46. Tong, H.; Wang, S.; Li, G. Credibility Based Imbalance Boosting Method for Software Defect Proneness Prediction. *Appl. Sci.* **2020**, *10*, 8059. [[CrossRef](#)]
47. Christakis, M.; Heizmann, M.; Mansur, M.N.; Schilling, C.; Wüstholtz, V. Semantic Fault Localization and Suspiciousness Ranking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019, Prague, Czech Republic, 6–11 April 2019*; Vojnar, T., Zhang, L., Eds.; Springer: Cham, Switzerland, 2019; Volume 11427. [[CrossRef](#)]
48. Hewett, R. Program Spectra Analysis with Theory of Evidence. *Adv. Softw. Eng.* **2012**, *2012*, 642983. [[CrossRef](#)]
49. Oberai, A.; Yuan, J.-S. Efficient Fault Localization and Failure Analysis Techniques for Improving IC Yield. *Electronics* **2018**, *7*, 28. [[CrossRef](#)]
50. Wong, W.E.; Debroy, V.; Li, Y.; Gao, R. Software Fault Localization Using DStar (D*). In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, Washington, DC, USA, 20–22 June 2012; pp. 21–30. [[CrossRef](#)]