SOFTWARE RELIABILITY ANALYSIS BY USING THE BIDIRECTIONAL ATTENTION BASED ZEILER-FERGUS CONVOLUTIONAL NEURAL NETWORK

D. Sudharson[*], *R. Gomathi*[†], *L. Selvam*[‡]

Abstract: Software quality assurance relies heavily on software reliability as one of its primary metrics. Numerous studies have been conducted to identify the software reliability. Improved software dependability may be studied using a triangular approach that includes software modeling, measurement, and improvement. Each of these steps is critical to the development of a solid software system. Improved accuracy in calculating dependability is critical to managing the quality of software. It has been discovered that deep learning algorithms are excellent methods of assessing many aspects of software dependability. Software systems contain distinct characteristics that can be addressed using deep learning techniques. In this study, a deep-learning-based bidirectional attention-based Zeiler-Fergus convolutional neural network (BA-ZFCNN) technique has been suggested to assess software dependability. In the beginning, the data were standardized by using the scalable error splash method. This approach was then used to extract the software fault-related characteristics using hypertuned evolutionary salp swarm optimization (HESSO). Finally, the Zeiler-Fergus convolutional neural network based on bidirectional attention (BA-ZFCNN) may be used to assess software dependability. The suggested method is used to forecast how many defects or failures there are in a software product. AR1 software defect data is widely used to test the effectiveness of deep learning and traditional machine learning methods. The experimental results reveal that the proposed method's accuracy (96.7%) is higher than the current techniques' accuracy.

Key words: *software reliability, scalable error splash technique, hypertuned evolutionary salp swarm optimization, bidirectional attention based Zeiler-Fergus convolutional neural network*

Received: December 6, 2022 **DOI:** 10.14311/NNW.2024.34.001
Revised and accepted: February 19, 2024

[*]Sudharson Dorai Samy – Corresponding author; Department of AI & DS, Kumaraguru College of Technology, Coimbatore, Tamilnadu, India, E-mail: dsudharsonresearch@gmail.com

[†]Gomathi Ramalingam; Department of Electronics and Communication Engineering, Anna University Regional Campus Coimbatore, Tamilnadu, India, E-mail: rgomathi@aurcc.ac.in

[‡]Selvam Lakshmanan; Department of CSE, Karpagam Academy of Higher Education, Coimbatore, Tamilnadu, India, E-mail: umaselvam_35@yahoo.com

1. Introduction

The dependability of the software is one of the most critical factors in the overall reliability of the system. Additionally, there is the possibility that software will function properly for a certain amount of time. Reliability is defined as the program's ability to fulfill its desired function under certain conditions over a set period of time in the IEEE Standard Glossary of Software Engineering Terminology (Standards Coordinating Committee of the IEEE Computer Society, 1991).

In order to have bug-free modules in the software, all of these steps must be properly synchronized. The mean time to failure of software reliability assessment using deep learning and cumulative fault count between intervals may be estimated with the use of SRGM. For software failure prediction, DL approaches have been shown to be more accurate than statistical methods in forecasting better outcomes. These solutions need considerably fewer assumptions and leverage previous failure data as input for software with complicated phenomena. Systems may learn and adapt on their own by using historical and current failure data to infer future system behavior using a technique called deep learning (DL).

Using cumulative data, we compared the accuracy of reliability prediction and found that the latter is always preferred over the former. Cumulative data show a strong positive linear correlation between the actual and expected values for reliability prediction when looking at the correlation coefficient. The contribution of the paper can be listed below: to classify software faults and continuously learn to improve the accuracy of deep learning. We first use hypertuned evolutionary salp swarm optimization to analyze the source code and isolate the faulty features.

In the classification part, we use bidirectional attention-based Zeiler-Fergus convolutional neural network for classifying the code faults to check the software reliability. Software fault localization and maintainability are defined as a software system or modules that could be adapted to correct faults, improve performance, system testing, use software development techniques, or be modified for a changed platform. A software defect predictive model enables organizations to reduce the maintenance effort, time, and overall cost of a software project. To ensure the quality of good software, it must be reliable, and can tolerate a smaller number of failures during the software's runtime.

Hence, the classification of defects in software modules has a large impact on the software development process. The real scenario would become hard when a developer changes his program inside an application and it is related to other modules, including the failure of the updated version of this application. Therefore, it is very possible for the software to become faulty and not be stable. The number of studies in software fault proneness is increasing day by day due to the demand for automated services that can be depicted in Section 2. Section 3 explains the problem statement and existing methodology for software prediction. The proposed software defect predictive development model using deep learning techniques to enable the software to continue its projected task is elaborated in Section 4. Moreover, we have used different prominent evaluation benchmarks to evaluate the model's performance, which is illustrated in Section 5.

2. Related Works

In this section, we provide research on software metrics, classification and regression models for defect detection, and deep learning and its applications.

2.1 Software Metrics

Nested stacking, heterogeneous feature selection [1], and a correlation-based modified long short-term memory network approach [2] have been frequently used in software fault prediction. Combinations of complicated measures may be used to anticipate errors, according to [3]. It was decided to make the Eclipse datasets accessible for anyone to utilize in developing defect prediction algorithms. A thorough review of software defect prediction studies using various metrics, approaches, and datasets was conducted in [4]. In the prior study on defect prediction, they discovered that measurements at the method level were mostly used as metrics. When it comes to defect detection, class-level metrics should be used more often since they may help forecast problems in the design phase of the project. In [5], the author illustrates a variety of indicators (such as source code churn, source code entropy, and process metrics) and develops a variety of classification models for defect prediction. In order to simplify defect prediction for unlabeled datasets, CLA and CLAMI [6] introduced two unique ways for labelling unlabeled datasets automatically using the magnitudes of metric values. Using process-based software metrics, [7] performed a trend analysis of early software defect prediction. Cross-project defect prediction was not handled by heterogeneous metric sets, hence, [8] presented heterogeneous defect prediction using matching metrics. More and more software metrics-based defect prediction algorithms have been suggested [9–13]. Supervised defect prediction methods and unsupervised defect prediction methods are two distinct categories of this sort of methodology. When developing a model for supervised defect prediction, historical datasets are used to train the algorithm. Software dependability growth models were constructed by using the pseudo inverse learning method and the stacking generalisation approach [14, 15]. Additionally, they used a support vector machine (SVM) to make predictions about software quality. A model for predicting software defects at the method level was developed by [16, 17] using an RF algorithm that was based on historical measurements at the method level. Unsupervised defect prediction methods may predict fault proneness without the need for a defect dataset. When a training dataset is either inadequate or unavailable, this method might be implemented. The raw value of each change metric's reciprocal [18] offered an unsupervised method for ranking the change metrics in decreasing order.

2.2 Software Defect and Reliability Prediction Classification Models

Classification and regression models may be used to assist engineers in detecting and fixing software faults more quickly and with less effort. The most frequent kind of fault prediction model is a classification model. Categorization models from the NASA metrics data collection were empirically compared on a wider

scale with 10 public datasets. [19] established a framework for comparing software defect classification predictions. [20] used a variety of classification approaches to create a variety of classification models, which the authors then evaluated on three distinct datasets. According to their findings, the performance of defect prediction models varies greatly across various classification methods. A total of 24 distinct classifier techniques were replicated [21] (e.g., logistic regression, C4.5 decision tree, RF, etc.). When it came to forecasting software that is prone to errors, the logistic regression model performed best. The just-in-time defect prediction technique for the prediction of commits that introduce defects and the identification of lines that are linked with these defects are described in [22] (i.e., defective lines). According to [23], the KTSVMs are proposed to execute domain adaptation (DA) in order to fit the training data distributions for various applications. Additionally, the CPDP model uses KTSVMs with DA functions (known as DA-KTSVMs). Using a dictionary learning method, [24] suggests that software problems may be anticipated. An unsupervised classifier, based on connectedness, was used to solve this problem [25]. [26] showed that fault classification was enhanced using feature selection and ensemble learning. Choosing your features with care is essential if we want reliable classification models. Caret, an automated parameter optimization tool, was used in the study of classifier model performance [27]. Classifiers trained using caret were found to be as stable as those trained using the default parameters and to perform better than those taught using just the default settings. Defect prediction models' performance may be greatly affected by parameter adjustments. To better understand how defect classification models function, [23] looked at how feature selection strategies affect their performance. Classification models for defect prediction should include feature selection strategies, they said. Defect predictors may be built both inside and between projects by employing a simpler metric set, as was shown in the study by [28]. Research by [29] was the first to use online change categorization to enhance fault prediction. Many composite methods integrating different machine learning classifiers for defect prediction were researched by [30] in order to increase the performance of cross-project defect prediction.

2.3 Software Reliability Prediction Using Regression Models

The use of regression models to predict defects has been studied less thoroughly than the use of categorization models to forecast defects. Regression models are used to identify possible problems in a systematic manner. Flaw-prone modules are ranked higher than those that are defect-free. The number of bugs in software modules is difficult to estimate with any degree of certainty. A proper regression model is essential in this case. To forecast software failures, [31] used a linear regression model on a change-level dataset. An effort-aware defect prediction model was developed using linear regression on a dataset of change measurements by the researchers at [32, 33].

The Alberg diagram may be used to quantify regression model performance. In order to enhance the ranking model's performance measure, [34] proposed a learning-to-rank technique. Module failure rates of 20% or more are used to assess models, as are fault-percentile averages (FPA) and module failure rates (MFR).

Defect acceleration predictor factors were used in a MACLI approach established in [35] to estimate the amount of defects in a prospective product release and to evaluate each predictor variable's connection with defects and there is a strong correlation between the average defect velocity and the actual number of defects. SVM was initially proposed in [36]. SVM's theoretical underpinnings are based on a concept known as structured risk minimization (SRM). The SVM was created to address classification difficulties.

Linear and nonlinear regression issues may be solved with the help of SVR, an extension of SVM. It was shown that utilising FSVR to predict the quantity of software defects improved performance by 7%. [37] Fuzziness in the regression approach may be used to deal with unbalanced datasets. C4.5 and RF decision trees were used to build a cost-sensitive software quality prediction model.

Using decision tree regression (DTR), [38, 39] a model for fault count was built (i.e., within- and cross-project defect scenarios). Predicting defects both inside and between projects was not a problem for the defect prediction algorithms that were tested. The DTR was used to anticipate defects both during and after release. [40] A recent empirical study [41] focused on defect prediction models that could anticipate the number of faults that would be found in a product. Based on average absolute error, average relative error, and level-l metrics, the DTR model was shown to be the most accurate prediction model.

2.4 Deep Learning and Its Applications

Deep learning approaches include convolutional neural networks (CNN), recurrent neural networks (RNN), and long-short term memory (LSTM). Deep learning has been used in speech recognition, natural language processing, and picture processing. Deep learning is increasingly being used by software developers. [42] discovered that the functional similarity of code may be assessed using a new approach called DeepSim. After embedding code in a matrix, they used a DNN model to learn features from the matrix and perform a binary classification. In [43], an autonomous neural network debugging approach called MODE was introduced. This method was used to identify problems in the models and to pick training inputs, like regression testing and software debugging. MODE has the ability to detect and rectify model faults very rapidly.

A feed-forward network and an RNN deep learning software language model were used to provide programming code suggestions. [44] Deep software language models were also used to predict software engineering activity. Deep learning algorithms may be applied to source code files to build high-quality models, as illustrated by the use of a Java project corpus. The salp swarm algorithm (SSA) and backpropagation neural network (BPNN) are used by [45].

A combination of lexical and programmatic structural information was used in the search for flaws so that unified features could be learned from both natural language and source code. They used CNN to get both comprehensive and semantic information. [48]'s deep learning method may be used to identify code clones. Using this strategy, the source code is automatically searched for unique features. There was a suggestion that all of the source code's phrases and fragments may be used to identify clones. In order to anticipate semantically connected knowledge

units, [49] proposed using deep learning. Instead of using binary classification, they used a multi-layer classification model. Knowledge units are represented by word embeddings and a CNN at the word and document level.

Compared to a naive Bayes model, the ensemble model performed substantially better, according to the authors. One ensemble model and a few NASA datasets were used by the authors. In [54], a similar study, three SFP cost-sensitive boosting strategies were presented and evaluated. According to the author's results, one threshold-updating and two weight-adjusting algorithms were used to analyse four NASA datasets. When it came to SFP, the approach based on the neural network's threshold update was found to be more effective. In [55], an investigation into the use of classifier ensembles to predict software flaws was described. As the primary learner among the ensemble techniques, bagging, boosting, random trees, and random forests were all examined together with stacking and voting, random subspace, and naive Bayes.

A series of tests, including several NASA datasets, showed that voting and random forest were superior to other alternatives. On average, ensemble techniques beat single classifiers. [56] developed a failure prediction model based on an ensemble method for a large-scale software system. According to the author, ensemble tactics based on decision trees outperformed other approaches and generated greater accuracy. The cost-sensitive analytic techniques CSForest and CSVoting were recently presented in [57]. To reduce the classification cost, the investigated ensemble approaches first constructed a collection of decision trees and then integrated these trees.

The level of knowledge needed by algorithms to categorize and choose the necessary quality characteristics for software fault prediction is usually low. This affects the overall efficiency of software fault prediction. Performance assessment measures are inconsistent with one another, which makes it difficult to determine which performance metrics to use. As a consequence, there is no information that can clearly characterize the application of any particular methodology to any particular sort of dataset in order to produce high performance outcomes. Thus, deep learning techniques are effectively utilized for assessing distinct characteristics of software systems. This novel approach will help to increase the efficiency of dependency detection, which in turn uncovers flaws in software that may lead to software defects or failures.

3. Proposed Methodology

Software reliability is an essential characteristic that is typically taken into account while trying to improve the quality of software. Reliability in software is concerned with the presence of flaws in the system. In most cases, a bug in the code is to blame for a system failure, although one bug might trigger a slew of them. The process of improving the system's quality by identifying and correcting flaws is known as "enhancing software dependability".

The dependability of software may be assessed using several analytical models, known as "software reliability growth models". LOC, a basic static code measure, has been shown to be an effective predictor of software problems. Furthermore, there is enough data to train any prediction model using these basic criteria. As

long as there are enough training models, it doesn't matter whether a simplified (or even minimal) feature subset works well both inside and between projects. The Ar1 dataset was obtained through publicly accessible embedded software written in the C programming language, as shown in Fig. 1.

The proposed methodology uses a preprocessing scheme where uncertainties in the dataset are determined by the error splash technique. Redundant risk data were identified and removed, and the dataset was normalized. The SSA and PSO algorithms are combined into a new algorithm known as hyper-tuned evolutionary salp swarm optimization (HESSO), which has been used for feature selection. Finally, we have used bidirectional attention-based Zeiler Fergus convolutional neural network (BA-ZFCNN) and HESSO to build a deep learning model that is capable of making accurate predictions.

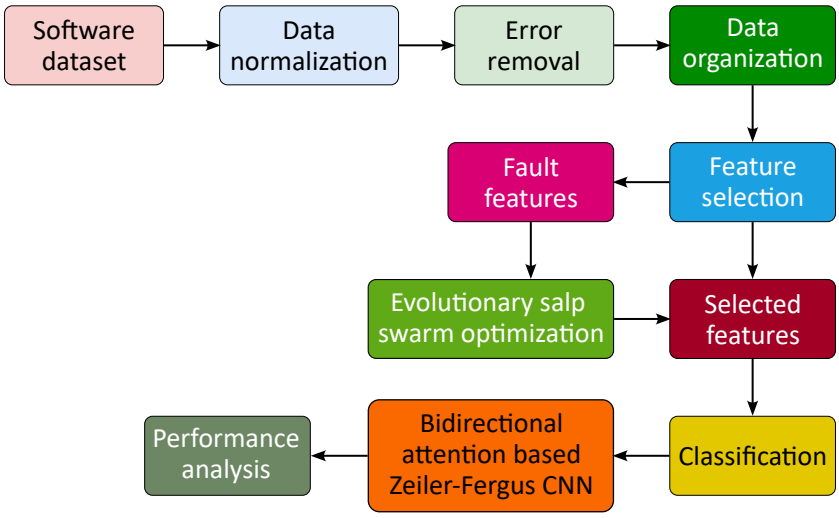

Fig. 1 *Schematic representation of the suggested architecture.*

3.1 Dataset

The proposed system uses the AR1 dataset, which is available as open source for testing the effectiveness of deep learning techniques. In this data, ar1.arff files have defect information in a discrete manner (whether there is a bug or not), whereas ar1_numericdefects.arff files have the bug count associated with the defectives.

3.2 Obtaining Spatial and Temporal Dependence

A preprocessing model calculated the uncertainty of the obtained data using the error splash technique. Estimating variance is done using the variance of the conditional distribution $N(k|v,w)$, where v and w represent events. The uncertainty is described using a Gaussian distribution N.

$$N(k \mid v,w) = N(k;\mu(v,w),\sigma^2(v,w)), \tag{1}$$

where $N(k \mid v, w)$ represents conditional probability of event k occurring given that events v and events w have occurred, $\mu(v, w)$ represents expected risk and $\sigma^2(v, w)$ is the variance.

Here $\mu(v, w)$ and $\sigma^2(v, w)$ whose expectation equals the calculated aleatoric uncertainty $\delta_x(k)$. Negative log-likelihood (NLL) was used as a loss function and the variance with respect to events v and w was calculated using the formula, $\sigma^2(v, w)$. As an indicator of the level of risk, an error count may be used. There have been L trainings of the network, each using a randomized training set, with random parameterization. Using uniform mixing model, L trainings of the networks were normalized by obtaining $\{e\}_{l=1}^{L}$, where e represents a set of events.

The variance of the expected error risk provides the uncertainty δ_e. Eqs. (2) and (3) may be used to get the final predictive uncertainty, which is equal to the mean of the projected errors.

$$\sigma^2(k) = \delta_x^2(k) + \delta_e^2(k), \tag{2}$$

$$\hat{\mu}(x) = \frac{1}{O} \sum_l \mu(x, w_l), \tag{3}$$

$$\hat{\delta}_x^2(k) = \frac{1}{O} \sum_l \sigma^2(x, w_l). \tag{4}$$

Here $\hat{\delta}_x^2$ represents the variance of a data population, $\hat{\mu}$ represents the overall mean of the data population.

It can be rewritten as follows:

$$\hat{\delta}_x^2(k) = \frac{1}{O} \sum_l \mu^2(x, w_l) - \hat{\mu}^2(x), \tag{5}$$

where $\hat{\delta}_x^2(k)$ represents variation or change, O is number of multiplicative observations, μ denotes a mean, x is a random variable, w_l represents different weights or factors.

There were distinct model parameters due to the differing scales $\hat{\delta}$ of in the network design. To address this issue, we proposed the $\mathbf{C}(a, k)$ confidence matrix.

$$\mathbf{C}(a, k) = N\left(k; \hat{\mu}(a), \hat{\delta}(k)\right) = \frac{1}{\sqrt{2\pi}\hat{\delta}(k)} e^{-\frac{(y-\hat{\mu}(a))^2}{2\hat{\delta}^2(b)}}, \tag{6}$$

where a, b are real numbers, y is the data coefficients (a, k) – probability density function which describes a normal (Gaussian) distribution N with a random event k being normally distributed around the mean $\hat{\mu}(a)$ with a standard deviation of $\hat{\delta}(k)$.

We propagate uncertainty by calculating the final quantity's probability distribution. To calculate the uncertainty of an expression directly, we can use the general form of summation in quadrature,

$$f(x, y, \ldots, n) = \sqrt{\left(\frac{\partial f}{\partial x}\delta x\right)^2 + \left(\frac{\partial f}{\partial y}\delta y\right)^2 + \cdots + \left(\frac{\partial f}{\partial n}\delta n\right)^2}, \tag{7}$$

where x, y are random variables, ∂f is the absolute error in $f(x, y, \ldots, n)$ resulting from errors $\delta x, \delta y, \ldots, \delta n$.

The following formula may be used to remove redundant risk data from the newly processed data.

$$Dup = \frac{Non - empty\, records}{Number\, of\, all\, records}, \tag{8}$$

$$Unq = 1 - \frac{Number\, of\, attribute\, duplicate\, value}{Number\, of\, all\, records}, \tag{9}$$

where *Dup* represents the identified duplicate records, *Unq* represents unique records.

$$Error\, value = \begin{bmatrix} 1 & 2 & 4 \\ 1/2 & 1 & 3 \\ 1/4 & 1/3 & 1 \\ 1/5 & 1/4 & 1/3 \\ 1/6 & 1/5 & 1/2 \end{bmatrix}. \tag{10}$$

Following is a description of how the error value is calculated,

$$Error\, value = 1 - \frac{Number\, of\, time - Series\, check\, error}{Total\, number\, of\, records}. \tag{11}$$

Finally, the errors in the dataset can be eliminated.

3.3 Feature Selection

In this part, the algorithm's structure is laid out. The SSA and PSO algorithms are combined in a new method known as HESSO. The SSA algorithm's fundamental structure is altered by enhancing the population's position update phase. The PSO's update mechanism is now part of the SSA's primary structure thanks to this change. This integration allows the SSA to explore the population with more freedom, increase its variety, and rapidly arrive at the ideal value. In general, the HESSO algorithm's primary component is shown in Fig. 2. First, the suggested HESSO generates a population that represents a collection of possible solutions to the issue at hand (feature selection). Once the fitness functions for all of the solutions have been computed, the best solution may be selected.

Once a fitness function has been established, the HESSO method's next step is to apply either the SSA or PSO algorithm to update the current population (measured by its probability). PSO and SSA are utilised if the fitness function has a probability greater than 0.5, respectively. An updated population of solutions is used to determine the optimum fitness function for each one, and the best one is then selected. Stop conditions are checked to verify whether they have been fulfilled before the optimum solution may be returned; if not, repeat the preceding stages from probability computation through completion. More information on each of these processes may be found in the sections that follow. First, the SSA and PSO parameters are set, and then the HESSO method uses these values to create a random population c of size M in dimension F, before using SSA to determine the food fitness of each solution c_o, $o = 1, 2, \ldots, M$.

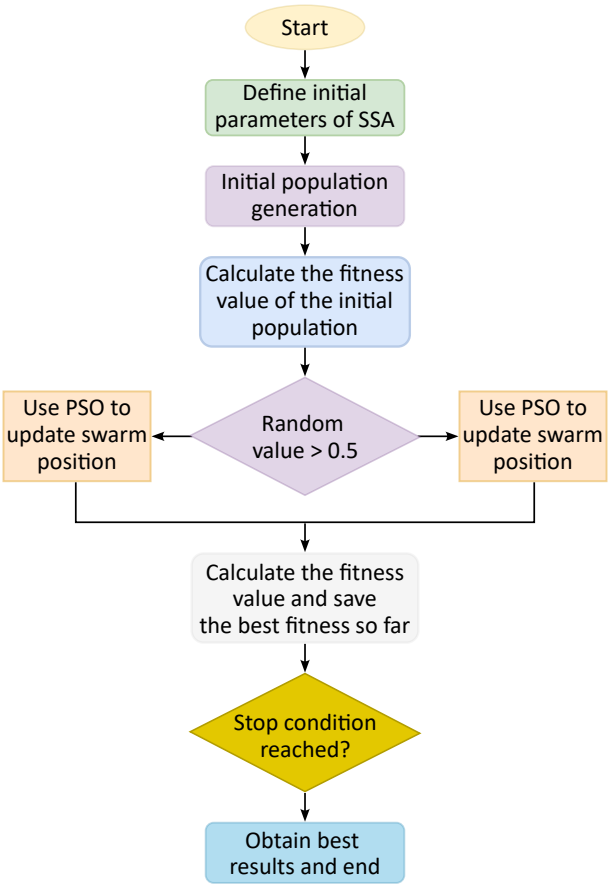

Fig. 2 *Process of optimization.*

The following equation is used to transform each solution c_o to a binary vector (consisting exclusively of 1 s and 0 s) based on a random threshold $V \in [0, 1]$ before calculating the objective function:

$$c_o(y+1) = \begin{cases} 1, & if \ \frac{1}{1+e^{-c_o(y)}} > V \\ 0, & otherwise, \end{cases} \tag{12}$$

represents the logistic function used for sigmoid activation function. It transforms the variable $c_o(y)$ using the logistic (sigmoid) function to produce a value between 0 and 1.

As a result, only c_o components with a value of 1 are used to reflect the chosen qualities (as well as the other components, which reflect unnecessary traits, are omitted). The goal function in Eq. (13) has now been calculated for each entity:

$$g(c_o(y)) = \xi R_{c_o(y)} + (1-\xi)\left(\frac{|c_o\,(y)|}{|v|}\right). \tag{13}$$

$g(c_o(y))$ – goal function of $c_o(y)$, $R_{c_o(y)}$ – number of features representing mistakes in data processing parameter $\xi \in [0,1]$ – used to balance between classification error and feature selection. For example, the parameter $\xi \in [0,1]$ is implemented in order to maintain a proper balance between classification error and feature selection. The probability of each fitness function Pro_o is then calculated as:

$$Pro_o = \frac{g_o}{\sum_{o=1}^{M} g_o}. \tag{14}$$

Current solution will be updated by either the SSA or PSO, depending on the Pro_o value.

Assume that $g_1, \ldots, g_u$ are observations, and that each observation is represented by an n-by-n row vector in order to calculate (the number of attributes). So, the dataset may be summarized in the form of the matrix $\mathbf{G}_{b\times n} = 1/(b \cdot g)$. The standard deviation of an observation from the average is given by the formula $\mu = \frac{1}{b}g$. The data set's sample covariance matrix is:

$$\mathbf{X} = \frac{1}{b}\sum_{b=1}^{n}(g_b - \mu)(g_b - \mu)^{\mathrm{T}}. \tag{15}$$

We choose eigenvectors with the largest eigenvalues. Within this "signal" subspace, only the first initial vectors of large dimensions are packed with information, while the rest are typically filled with noise. The following formula may be used to quickly determine how many dimensions k has:

$$k = \frac{\sum_{u=1}^{j}\lambda_u}{\sum_{u=1}^{n}\lambda_u}, \tag{16}$$

where λ is the ratio between the subspace's variation and the overall space's variation. The j eigenvectors are represented as columns in a $(n \times j)$ (usually $j \ll n$ for data reduction) of matrix $\mathbf{Y}$. According to ESSO's guidelines, data is projected onto the j-dimensional subspace as follows:

$$T_u = (Z_u - \mu)\mathbf{Y} = \phi\mathbf{Y}. \tag{17}$$

Here, T_u represents the variation in the subspace, ϕ is subspace constant, and Z_u represents overall variation observed with the variance of mean μ.

3.4 Classification

There are three layers of the network in the proposed model BA-ZFCNN: the network is made up of two step mappings, which may be recast as nested functions. Here, the input and output layer nodes are all linked to the hidden layer. 1) In the feed forward network, feedback from output nodes is ignored. These are the two most essential phases of feed forward neural network architecture. 2) Error propagation from the output layer is back propagated. The input vector is transferred from the input layer to the hidden layer, as shown in Fig. 3.

For each pair of input patterns, the error is backpropagated. As a result, the weights have been revised every time a mistake is found. There are nodes y_j that receive external inputs after the failure interval of $[j,1]$. Let's assume that $y_j = 0$

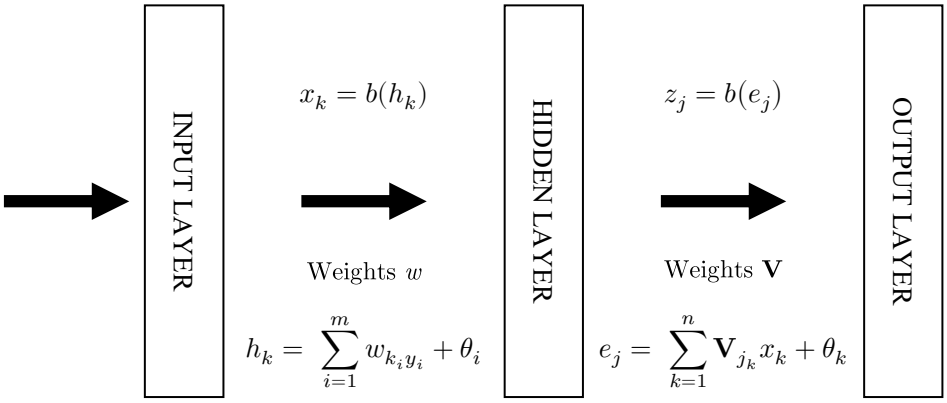

Fig. 3 *A simple architecture of feed-forward neural network.*

for nodes that aren't inputs and c_j is the intended output for the jth unit's desired state. Eq. (18) is used to calculate the hidden layer's output after accepting the net input.

$$x_k = b(h_k),$$

where

$$h_k = \sum_{i=1}^{m} w_{k_i y_i} + \theta_i, \tag{18}$$

where m is the the number of input nodes, w_{k_i} is the weight link between the ith input layer and the kth hidden layer node, and b is an activation function that must be continuous, differentiable, and not decrease over layers. The total number of cumulative failures is calculated using Eq. (19). The updated weight $\mathbf{V}_{jk}$ is computed for each network copy summated with inputs. After that, the weights are fine-tuned one by one. e_j describes the error function.

$$e_j = \sum_{k=1}^{n} \mathbf{V}_{j_k} x_k + \theta_{k.} \tag{19}$$

The number of "hidden" nodes is indicated by the letter n in the formula above. $\mathbf{V}_{jk}$ is the weight matrix, and θ_k is the bias (connecting kth to jth nodes).

Assume that the networks can be trained using j data points. Initial findings are based on observations made up to $(j+1)$ data points prior to this step. The following is the order in which the data on software failure time is created for training purposes. Eq. (20) must be applied to the cost function to calculate the MSE (mean square error).

$$MSE = \frac{1}{n} \sum_{l=1}^{n} (c_l - z_l)^2 = \frac{1}{n} \sum_{l=1}^{n} M_l^2. \tag{20}$$

Here,

$$M_l = \begin{cases} c_l - z_l, & for\, lth\, output\, node, \\ 0, & otherwise. \end{cases}$$

There are nodes in M_l that represent the summation ranges of all output units as a training sequence matrix index. c_l represents the desired cumulative failures, while z_l represents the predicted failures. The formula in Eq. (21) explains how to minimize the total error while training the network. Weight change gradients may be recorded using Eqs. (21) and (22).

$$\Delta v_{jk} = \eta(c_j - z_j)b^{'}(e_j)x_k, \tag{21}$$

$$\Delta w_{ki} = \eta \sum_{j=1}^{n} \left[(c_j - z_j)\, b^{'}(e_j)v_{jk}\right] b^{'}(h_k)y_i, \tag{22}$$

where η determines how quickly adjustments are made; this is known as the "learning rate" scalar parameter and b' is the derivative of b. h_k defines the learning parameter.

Algorithm 1 Classification process model.

Step 1: Data initialization
Set the $\mathbf{X}$ and $\mathbf{Y}$ weight matrices and the neuronal thresholds to values within the range $(0, \ldots, 0.5)$. In this case, the number of patterns and the error are both set to zero.
Step 2: Set tolerable error
Specify the maximum amount of mistake that may be tolerated $\mathbf{F}_{\max}$. Consider the precision of the error as $\mathbf{F}_{\max} = 0.005$.
Step 3: Provide input
Pairs $\{y(1), c(1)\}, \{y(2), c(2)\}, \ldots, \{y(q), c(q)\}$ may be used to feed the input. The cumulative execution time is represented by $y(j)$, while the target failure number is represented by $c(j)$.
Step 4: Exercise routine
To the input layer y, apply the lth input pattern (l).
Step 5: Propagation
Calculate the expected outputs $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n$ (i.e., next failure number)
Step 6: Calculate error
Equation is used to calculate the errors $\mathbf{M}_j$ for each output node $\mathbf{z}_j$
Step 7: Reverse error weight updating and propagation
The weight of faulty code should be updated.
Step 8: Manage the training environment
if $\mathbf{l} = \mathbf{q}$ **then**
go to Step 9;
else
update $\mathbf{l} = \mathbf{l} + 1$ and proceed to Step 4;
end if
Step 9: Compare errors
if $\mathbf{F}_{\max}$ is true **then**
go to Step 10;
else
put $\mathbf{M} = 0$, $\mathbf{l} = 1$, and proceed to Step 4;
end if

Step 10: Print the final weights value analysis
Step 11: Calculate the result
Calculate the failures that follow.
Step 12: Predictive measures computation.

The network parameters and layer weights are built up in Step 1 of our method. The maximum permissible MSE value is established in step 2. The inputs to this network are organized in Step 3. In Step 4, the network is trained, and in Step 5, the network's output is computed. The MSE is defined in Step 6 and used to compare predicted and actual error levels. The weights must be calculated and adjusted until the MSE value falls below a particular threshold. Based on this halting condition, the network is propagated back to the hidden layer. Back-propagation learning is used to train the network in Step 7, and the weights are adjusted accordingly. To guarantee that the process is functioning smoothly, the maximum allowed error is compared to the MSE after each epoch.

At this stage, the final tally is calculated. At the end of the procedure, the final weights are established and the output is recorded for the next cumulative failure in Step 11. All of the prediction criteria in Step 12 are calculated using formulas. The buried layer is thought to have a huge number of neurons. Even if the number of hidden layers is as large as intended, the findings show that increasing the number of hidden layers has no discernible effect on performance. Rather, the training causes a decrease in performance. Neurons in the input layer are not subjected to error calculation. Starting with a random set of weights and adjusting them in proportion to their contribution to inaccuracy in the first period is usual practice.

The error is estimated in the output layer, and the difference between the actual and intended output values is calculated. For cross-validation, the whole representative data set is split into two sets: a training set for training the network and a test set for forecasting the system's dependability. The data set was partitioned in the following way: 80% of the cash is spent on training, whereas only 20% is spent on testing. The data sets are normalized to their maximum values before being analyzed. When the interval is closer to [0,1], the model performs better for all data sets. The training inputs to the network and the network's target outputs make up the training pair.

For the training data, there are two-dimensional arrays (input-output): (J_1, P_1), $(J_2,\ P_2), \ldots, (J_j, P_j), \ldots,$ and $(J_m,\ P_m)$, where J_j and P_j represent the input values (i.e., cumulative execution time) and intended output values, respectively, in the form of a two-dimensional array: $(J_1,\ P_1)$. The number of failures may be calculated using the entire execution time. The weights are stored in two-dimensional data arrays (i.e., input-to-hidden layer and hidden-to-output layer). The logistic function used in this work is the binomial sigmoidal $g(y) = 1/(1+e-y)$, where y is the cumulative execution time. The binary sigmoidal function may be used to reduce the computational weight of training. $g(y)$ is a monotonically increasing function. At each learning level, the steepness parameter is shown. This logistic function may be used with a range of zero to one.

To project the incoming vector $\mathbf{r}$, which represents a test sample, the j-dimensional subspace of normal behavior was pointed. The distance is the difference between the mean-adjusted input data vector $\Phi = \mathbf{r} - \mu$ and the test data vector's subspace reconstruction.

$$\Phi_t = (\mathbf{r} - \mu)\, YY^{\mathbf{r}} = \Phi YY^{\mathbf{r}} \tag{23}$$

If the test data vector is normal, that is, if the test data vector is very near to the training vectors corresponding to normal behavior, the gap between the test data vector and its reconstruction will be low. In the experiments, the distance between these two vectors is calculated using the squared Euclidean distance:

$$\varepsilon = \|\Phi - \Phi_e\|^2 . \tag{24}$$

The anomaly index ε is a measure of the degree to which something is out of the ordinary. Φ and Φ_e defines the data output model parameter with numerical index e. If it falls below a specific threshold, the vector is deemed normal. If it is not, it is considered as an outlier.

3.4.1 Case of High Severity of Detected Fault

When a high-severity problem is discovered, the reliability growth model may be modeled as follows:

$$\frac{\mathrm{d}}{\mathrm{d}r} n_{s_1}(r) = a_1(r)\left[q_1(r) - n_{s_1}(r)\right], \tag{25}$$

where

$$a_1(r) = v_1(r)\left(\delta_1(r) + \sigma_1\gamma(r)\right). \tag{26}$$

$n_{s_1}(r)$ is the cumulative number of high severity faults identified, $q_1(r)$ defines the software's fault content function, and $a_1(r)$ provides the proportionality factor, which has been explicitly stated as the product of $\delta_1(r)$ and $v_1(r)$ in Eq. (25). For high-severity faults, the rate of $\delta_1(r)$ corresponds to the severity rate, and the rate of $v_1(r)$ relates to the rate of fault detection in this specific scenario. The severity of a defect is based on the premise that it is random. According to Eq. (25), a normalized Gaussian white noise represents the rate of fault severity High severity failures may lead to unpredictable swings in the system's output. $q(r)$ is defined as follows: $q_1(r) = qw_1 + \alpha_1 n_{s_1}(r)$, where α_1 represents the initial number of software faults, w_1 represents the fraction of high severity software faults, and n_{s_1} refers to the rate at which new software faults are added.

3.4.2 Case of Low Severity of Detected Fault

If a failure of low severity is identified, the reliability growth model may be described as follows based on assumptions:

$$\frac{\mathrm{d}}{\mathrm{d}r} n_{s_2}(r) = a_2(r)\left[q_2(r) - n_{s_2}(r)\right], \tag{27}$$

where

$$a_{2(r)} = v_2(r)\left(\delta_2(r) + \alpha_2\gamma(r)\right). \tag{28}$$

For example, in Eq. (27) above, the cumulative number of low severity faults detected is equal to $n_{s_2}(r)$, and the fault content function for low severity faults is defined by $q_2(r)$. In addition, $a_2(r)$ specifies the proportionality factor, which is

explicitly defined in Eq. (27) as the product of $\delta_2(r)$ and $v_2(r)$. Eq. (28) shows that $\delta_2(r)$ is used to describe the severity of low-severity defects, whereas $v_2(r)$ reflects the rate at which these defects are detected. The assumption states that the severity of a defect is randomized. As a result, in Eq. (28), the rate of fault severity relates to a standardized Gaussian white noise $\delta_2(r)$. In case of mild severity faults, δ_2 indicating the amount of erratic fluctuations $q_2(r)$ may be calculated using the following equation: $q_2(r) = qw_2 + \alpha_2 n_{s_2}(r) = a_2(r)\,[q_2(r) - n_{s_2}(r)]$, where w_2 indicates the percentage of low-severity faults in the program, and n_{s_2} refers to the rate at which faults are introduced into the system.

If we assume that $v_1(r)$ is the constant function, then the detection functions for high and low severity faults (i.e., $v_1(r)$ and $v_2(r)$ are both constant functions). In addition, it is assumed that $V_1(r)$ has a value of 1 and $V_2(r)$ that has a value of 2. In $x_1(r)$, the unpredictability is related to $v_1(r)$ As a result, substituting 1 for 3 in $x_1(r)$ is a rational way to reduce the amount of unpredictability. Furthermore, unpredictability in $x_2(r)$ may be traced back to $X_2(r)$. This means that 4 may be substituted with 2 as the magnitude of irregular fluctuations with $x_2(r)$. For the full model, the total number of errors is calculated as $n_s(r) = \sum_{u=1}^{2} n_{s_u}(r)$ and that for fixed errors as n_{s_u} as $n_x(r) = \sum_{u=1}^{2} n_{x_u}(r)$.

Thereby $n_{\mathrm{s}}(\mathrm{r})$ takes the form as follows:

$$\begin{aligned} n_s(r) &= \frac{qw_1}{1-\alpha_1}\left\{1 - e^{-\left((1-\alpha_1)v\delta_1 r - \left(\frac{1}{2}v^2\sigma_1^2 r\right)\right)}\right\} \\ &+ \frac{qw_2}{1-\alpha_2}\left\{1 - e^{-\left((1-\alpha_2)v\delta_2 r - \left(\frac{1}{2}v^2\sigma_2^2 r\right)\right)}\right\}, \end{aligned} \tag{29}$$

and $n_{\mathrm{x}}(r)$ assumes the form

$$\begin{aligned} n_x(r) &= \left(\left(\frac{qw_1}{x_1\,(1-\alpha_1)}\right)\left(1 - e^{-\left((1-\alpha_1)v\delta_1 r - \left(\frac{1}{2}v^2\sigma_1^2 r\right)\right)}\right)\right. \\ &\quad \left.\left(1 - e^{-\left(\left(\frac{e_1}{\delta_1}r\right) - \left(\frac{1}{2}\sigma_1^2 r\right)\right)}\right)\right) \\ &+ \left(\left(\frac{qw_2}{x_2\,(1-\alpha_2)}\right)\left(1 - e^{-\left((1-\alpha_2)v\delta_2 r - \left(\frac{1}{2}v^2\sigma_2^2 r\right)\right)}\right)\right) \\ &+ \left(1 - e^{-\left(\left(\frac{e_2}{\delta_2}r\right) - \left(\frac{1}{2}\sigma_2^2 r\right)\right)}\right). \end{aligned} \tag{30}$$

The section above wraps up the model building framework with providing assumptions for the model formulation. Finally, the fault of high and low severity can be classified.

Algorithm 2 BA-ZFCNN.

Input: **d** – processed dataset, **l** – dataset true labels,
Output: classified code
Initialise the swarm of salps $\mathbf{z}_u$ $(u = 1, 2, \ldots, N)$
Evaluate the fitness of each salp
Select the best salp $\mathbf{z}_u$ from the salp swarm
Initialize the maximum number of iterations $r_{\max}$ and loop counter $r = 0$

Main loop:
while $(r < t_{\max})$ **do**
  for each salp $\mathbf{z}_u$ **do**
    if $\mathbf{g}_u \sim$ leading salp **then**
      update the position of leading salp
    else if $\mathbf{g}_u \sim$ followers **then**
      update the position of leading salp
    end if
    check and repair salp swarm if crossed the search boundaries
    update the best salp
    $r = t + 1$
  end for
end while
let $\mathbf{f}$ be the feature set
for i in dataset **do**
  let $\mathbf{f}_i$ be the matrix features
  for j in i **do**
    $\mathbf{v}_j \leftarrow$ value vectorization $(\mathbf{j}, \mathbf{w})$
    alter $\mathbf{v_j}$ to $\mathbf{f_j}$
    adjust $\mathbf{f_i}$ to $\mathbf{f}$
    $\mathbf{f}_{\text{train}}, \mathbf{f}_{\text{test}}, \mathbf{I}_{\text{train}}, \mathbf{I}_{\text{test}} \leftarrow$ split features and train it
    $\mathbf{M} \leftarrow \text{ZFCNN}(\mathbf{f}_{\text{train}}, \mathbf{I}_{\text{train}})$
    Score $\leftarrow$ evaluation $(i, \mathbf{I}_{\text{test}}, \mathbf{M})$
    return
    score value
    classify features (test)
  end for
end for

4. Results and Discussion

It is difficult to measure the trustworthiness of current software since it is produced in a variety of sizes and functionalities.. We developed the HESSO BA-ZFCNN model using deep learning which can reliably forecast outcomes. This deep learning model can adapt to capture the training parameters of a given dataset as well as deepen the layer levels. A detailed examination and feature extraction demonstrate the model's potential for prediction. A deep learning model is used in this work to estimate software dependability and forecast the number of errors in the program.

Classification over dangerous software codes and trustworthy codes projected rates is shown in Fig. 4. Code characteristics were ranked before categorization in order to determine the most significant elements. The HESSO BA-ZFCNN model may also anticipate dangerous code, depending on the trust rank value. The actual positive and negative readings will show how much positive prediction there is in a given situation. The proposed method's positive prediction rate is higher than the genuine negative rate in this case.

Fig. 5 shows that the proposed detection model varies from the actual dataset. There was a considerable discrepancy in the suggested method for fault prediction.

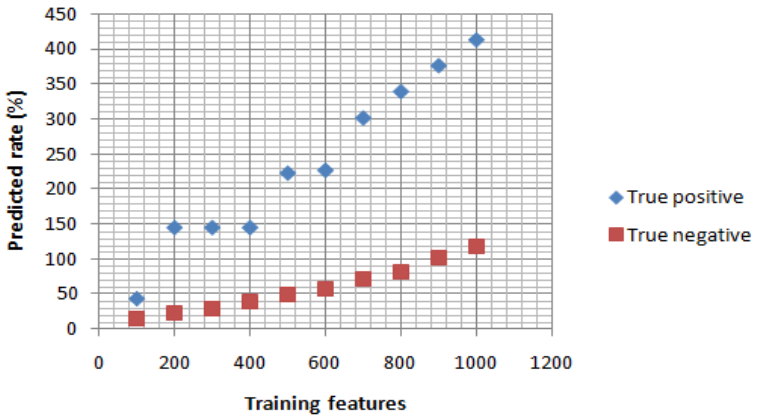

Fig. 4 *Training features vs. predicted rate.*

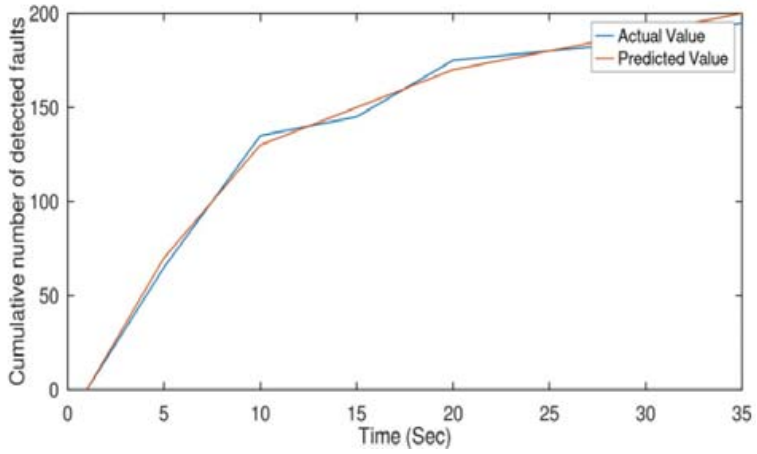

Fig. 5 *Time vs. cumulative number of detected faults.*

So many evaluation criteria are being used in this research, including specificity and accuracy as well as the area under the receiver operating characteristic (ROC) or AUC curve. When evaluating a predictor or classifier, the default cut-off value for the estimated probability of defect occurrences is not the required cut-off value of 0.5. Each of the following evaluation standards has been met:

$$Specificity = \frac{TN}{TN + FP}, \tag{31}$$

$$Sensitivity = \frac{TP}{TP + FN}, \tag{32}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{33}$$

$$Error\,rate = 1 - \frac{TP + TN}{TP + TN + FP + FN}. \tag{34}$$

Here TP represents number of true positives, FN represents number of false negatives, TN represents number of true negatives, FP represents false positives.

The number of software codes that can be correctly classified may be calculated. It determines how close the results are to what was predicted. Divide the total of actual positives and true negatives by the number of projected positives and negatives. In contrast, the proposed method's accuracy (96.7%) is higher than the current techniques' accuracy (see Fig. 6).

The extent to which a classifier can accurately identify all codes that do not have an issue with their condition is referred to as their specificity. In this case, as shown in Fig. 7, the recommended procedures had a large range of specificity (90%), which was quite high in comparison to other already existing mechanisms.

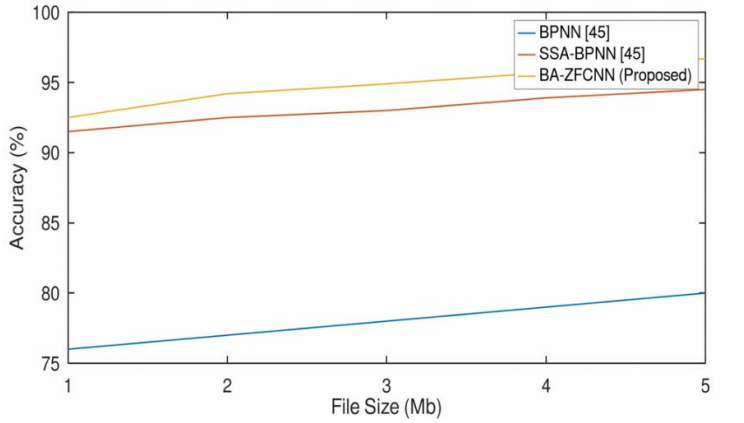

Fig. 6 *Comparison of accuracy with traditional and proposed method.*

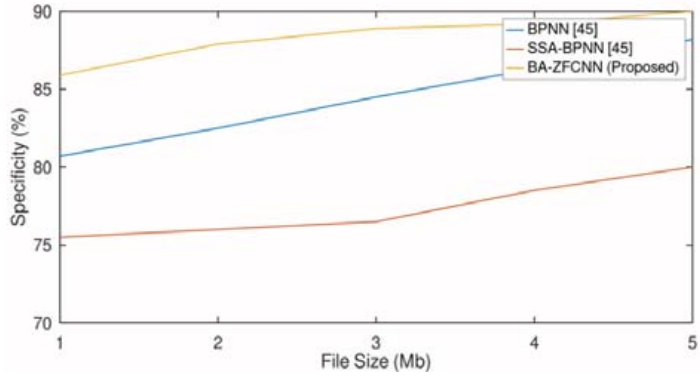

Fig. 7 *File size vs. specificity.*

The faulty condition prediction capability of a proposed system is shown graphically by an AUC curve. When compared to other existing mechanisms, the AUC range of the proposed mechanism has a very high AUC (99%), as shown in Fig. 8.

To correctly determine all codes with a faulty condition, or if 100% accurate, to identify all codes with a faulty condition using BA-ZFCNN. Fig. 9 shows that the proposed approach's sensitivity (100%) was quite high when compared to other existing mechanisms.

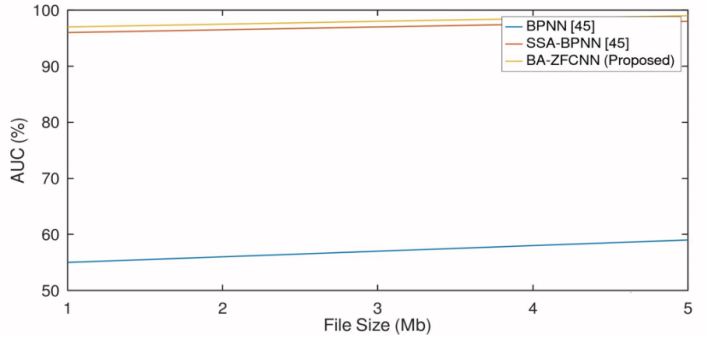

Fig. 8 *File size vs. AUC.*

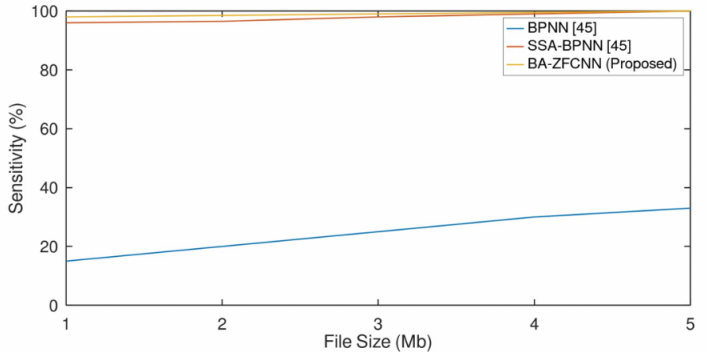

Fig. 9 *File size vs. sensitivity.*

Fig. 10 shows the error rate was less (0.03%) for the suggested methodology when compared to other existing mechanisms.

From the results of the above analysis, it was revealed that the suggested methodology expresses better results than other existing mechanisms. The deep learning technique BA-ZFCNN proposed in this research work can be used by the software developers to determine the dependability of the software systems, which leads to software defects or failures more accurately. Hence, this work can serve as a vital tool in software reliability performance assessment. Therefore, software defects and failures can be effectively minimized and reliability can be achieved with optimal efforts.

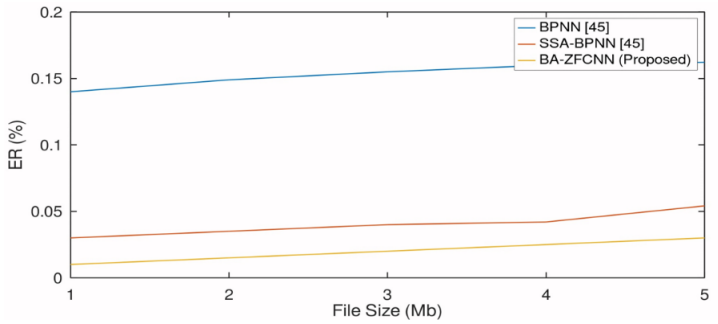

Fig. 10 *File size vs. ER.*

Model for release time computation	Optimal time T^* to release software (week)	Reliability achieved at T^* (week)	Cost of development at T^* (dollar)	AE	RE
IT2 FN-based model [46]	63.27	0.82	506,919.00	10.27	0.19
Optimal release time determination using type-1 fuzzy sets (Kapur et al. 2011b) [46]	0.12×10^{-3}	0.12×10^{-4}	141,393.70	52.99	0.99
Bi-criteria release time determination problem (Kapur et al. 1994) [46]	22.00	0.80	165,432.60	31.00	0.58
Proposed	30.00	0.87	32,788.90	9.55	0.01

Tab. I *Performability comparison, absolute error (AB), relative error (RE).*

5. Conclusion

Using the HESSO and the BA-ZFCNN, this research provided a solution to software reliability prediction difficulties. HESSO is used to determine the best BPNN parameters in this hybrid model, resulting in improved prediction accuracy. BA-ZFCNN was tested on a range of software failure prediction issue datasets to see whether it was effective. The PROMISE repository provided these datasets. AUC, sensitivity, specificity, accuracy, and error rate were used as performance indicators to assess the proposed method. HESSO with the BA-ZFCNN outperformed all other available techniques. Data sets and performance measures were all beaten by this algorithm. There are several advantages of using BA-ZFCNN as a tool for tackling software engineering challenges, including better prediction accuracy for a broad variety of SFP problems, as well as better resolution and lower error rates

than previous SFP approaches. SFP challenge success has led to the recommendation of the BA-ZFCNN for use in further prediction issues. For most datasets, the proposed method needs a significant amount of processing time. Consequently, future studies might create a new technique for optimizing the algorithm's computing cost.

References

[1] CHEN L.Q., WANG C., SONG S.-L. Software defect prediction based on nested-stacking and heterogeneous feature selection, Complex & Intelligent Systems, 2022, pp. 1–16.

[2] PEMMADA S.K., BEHERA H., NAYAK J., NAIK B. Correlation-based modified long short-term memory network approach for software defect prediction, Evolving Systems, pp. 1–19, 2022.

[3] QIAO L., LI X., UMER Q., GUO P. Deep learning based software defect prediction, Neurocomputing, 2020, 385, pp. 100–110.

[4] AKMEL F., BIRIHANU E., SIRAJ B. A literature review study of software defect prediction using machine learning techniques, Int. J. Emerg. Res. Manag. Technol, 2017, 6, pp. 300–306.

[5] ULAN M., LÖWE W., ERICSSON M., WINGKVIST A. Weighted software metrics aggregation and its application to defect prediction, Empirical Software Engineering, 2021, 26, pp. 1–34.

[6] NAM J., KIM S. Clami: Defect prediction on unlabeled datasets (t). In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 452–463.

[7] PANDEY S.K., MISHRA R.B., TRIPATHI A.K. Machine learning based methods for software fault prediction: A survey, Expert Systems with Applications, 2021, 172, p. 114595.

[8] ZHONG Y., SONG K., LV S., HE P. An Empirical Study of Software Metrics Diversity for Cross-Project Defect Prediction, Mathematical Problems in Engineering, 2021.

[9] FENG S., KEUNG J., YU X., XIAO Y., ZHANG M. Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction, Information and Software Technology, 2021, 139, p. 106662.

[10] FENG S., KEUNG J., YU X., XIAO Y., BENNIN K.E., KABIR M.A. COSTE: Complexity-based Over Sampling Technique to alleviate the class imbalance problem in software defect prediction, Information and Software Technology, 2021, 129, p. 106432.

[11] ZHU K., YING S., ZHANGN., ZHU D. Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network, Journal of Systems and Software, 2021, 180, p. 111026.

[12] BALOGUN A.O., BASRI S., MAHAMAD S., ABDULKADIR S.J., CAPRETZ L.F., IMAM A.A. Empirical analysis of rank aggregation-based multi-filter feature selection methods in software defect prediction, Electronics, 2021, 10, p. 179.

[13] SINGH P.D., CHUG A. Software defect prediction analysis using machine learning algorithms. In: 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence, 2017, pp. 775–781.

[14] MALHOTRA R. A systematic review of machine learning techniques for software fault prediction, Applied Soft Computing, 2015, 27, pp. 504–518.

[15] SAHU K., ALZAHRANI F.A., SRIVASTAVA R., KUMAR R. Evaluating the impact of prediction techniques: Software reliability perspective, Comput. Mater. Continua, 2021, 67, pp. 1471–1488.

[16] LONGRAIS G.P. Development of a Software Reliability Prediction Method for Onboard European Train Control System, ed., 2021.

[17] EKEN B., TOSUN A. Investigating the performance of personalized models for software defect prediction, Journal of Systems and Software, 2021, 181, p. 111038.

[18] ALBAHLI S, A deep ensemble learning method for effort-aware just-in-time defect prediction, Future Internet, 2019, 11, p. 246.

[19] KAUR A., KAUR I. An empirical evaluation of classification algorithms for fault prediction in open source projects, Journal of King Saud University-Computer and Information Sciences, 2018, 30, pp. 2–17.

[20] XU Z., LI L., YAN M., LIU J., LUO X., GRUNDY J., ZHANG Y., ZHANG X. A comprehensive comparative study of clustering-based unsupervised defect prediction models. Journal of Systems and Software, 2021, 172, p.110862.

[21] MOUSSA R., AZAR D., SARRO F. Investigating the Use of One-Class Support Vector Machine for Software Defect Prediction, arXiv preprint arXiv:2202.12074, 2022.

[22] FENG S., KEUNG J., ZHANG P., XIAO Y., ZHANG M. The impact of the distance metric and measure on SMOTE-based techniques in software defect prediction, Information and Software Technology, 2022, 142, p. 106742.

[23] RAHMAN M.H., SHARMIN S., ISLAM M.S., KHALED S.M., SARWAR S.M. An Attribute Selection Process for Cross-Project Software Defect Prediction. DUJASE, 2021, 6(1) pp. 6–15.

[24] WANG S. Leveraging Machine Learning to Improve Software Reliability, 2019.

[25] CHEN X., ZHANG D., ZHAO Y., CUI Z., NI C. Software defect number prediction: Unsupervised vs supervised methods, Information and Software Technology, 2019, 106, pp. 161–181.

[26] LARADJI I.H., ALSHAYEB M., GHOUTI L. Software defect prediction using ensemble learning on selected features, Information and Software Technology, 2015, 58, pp. 388–402.

[27] KUMAR R., CHATURVEDI A., KAILASAM L. An Unsupervised Software Fault Prediction Approach Using Threshold Derivation, IEEE Transactions on Reliability, 2022, 71(2), pp. 911–932, doi: 10.1109/TR.2022.3151125.

[28] PORNPRASIT C., TANTITHAMTHAVORN C.K. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 369–379.

[29] EKEN B., TUFAN S., TUNABOYLU A., GULER T., ATAR R., TOSUN A. Deployment of a change-level software defect prediction solution into an industrial setting, Journal of Software: Evolution and Process, 2021, 33, p. e2381.

[30] CHALLAGULLA V.U.B., BASTANI F.B., YEN I.-L., PAUL R.A. Empirical assessment of machine learning based software defect prediction techniques, International Journal on Artificial Intelligence Tools, 2008, 17, pp. 389–400.

[31] KAMEI Y., SHIHAB E., ADAMS B., HASSAN A.E., MOCKUS A., SINHA A., UBAYASHI N. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 2012, 39(6), pp. 757–773.

[32] KAMEI Y., MATSUMOTO S., MONDEN A., MATSUMOTO K.-I., ADAMS B., HASSAN A.E. Revisiting common bug prediction findings using effort-aware models. In: 2010 IEEE international conference on software maintenance, 2010, pp. 1–10.

[33] YANG X., WEN W. Ridge and lasso regression models for cross-version defect prediction, IEEE Transactions on Reliability, 2018, 67, pp. 885–896.

[34] YU X., KEUNG J., XIAO Y., FENG S., LI F., DAI H. Predicting the precise number of software defects: Are we there yet?, Information and Software Technology, p. 106847, 2022.

[35] KIRAN N.R., RAVI V. Software reliability prediction by soft computing techniques, Journal of Systems and Software, 2008, 81, pp. 576–583.

[36] DAS CHAGAS MOURA M. , ZIO E., LINS I.D., DROGUETT E. Failure and reliability prediction by support vector machines regression of time series data, Reliability Engineering & System Safety, 2011, 96, pp. 1527–1534.

[37] MAHATO S., DIXIT A.R., AGRAWAL R. Development of a Mathematical Model for the Software Defect Rework Process to Optimize Defect Rework—A Six-Sigma Case Study, in Recent Advances in Industrial Production, ed: Springer, 2022, pp. 403–410.

[38] PACHOULY J., AHIRRAO S., KOTECHA K., SELVACHANDRAN G., ABRAHAM A. A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools, Engineering Applications of Artificial Intelligence, 2022, 111, p. 104773.

[39] RATHORE S.S., KUMAR S. A decision tree regression based approach for the number of software faults prediction, ACM SIGSOFT software engineering notes, 41, pp. 1–6, 2016.

[40] RATHORE S.S., KUMAR S. An empirical study of some software fault prediction techniques for the number of faults prediction, Soft Computing, 21, pp. 7417–7434, 2017.

[41] RANJAN P., KUMAR S., KUMAR U. Software fault prediction using computational intelligence techniques: A survey, Indian Journal of Science and Technology, 2017, 10, pp. 1–9.

[42] ZHAO G., HUANG J. Deepsim: deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 141–151.

[43] MA S., LIU Y., LEE W.-C., ZHANG X., GRAMA A. MODE: automated neural network model debugging via state differential analysis and input selection. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 175–186.

[44] LE T.H., CHEN H., BABAR M.A. Deep learning for source code modeling and generation: Models, applications, and challenges, ACM Computing Surveys (CSUR), 2020, 53, pp. 1–38.

[45] KASSAYMEH S., ABDULLAH S., AL-BETAR M.A., ALWESHAH M. Salp swarm optimizer for modeling the software fault prediction problem, Journal of King Saud University-Computer and Information Sciences, 2022, 34(6), pp. 3365–3378, doi: 10.1016/j.jksuci.2021.01.015.

[46] CHATTERJEE S., CHAUDHURI B., BHAR C. Optimal release time determination via fuzzy goal programming approach for SDE-based software reliability growth model, Soft Computing, 2021, 25, pp. 3545–3564.

[47] XIAO Y., KEUNG J., MI Q., BENNIN K.E. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering, 2018, pp. 101–111.

[48] WARTSCHINSKI L., NOLLER Y., VOGEL T., KEHRER T., GRUNSKE L. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python, Information and Software Technology, 2022, p. 106809.

[49] ZHAO Y., SU T., LIU Y., ZHENG W., WU X., KAVULURU R., YU T. Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps. ACM Transactions on Software Engineering and Methodology (TOSEM), 2022, 31(3), pp. 1–33.

[50] WU B., LIANG B., ZHANG X. Turn Tree into Graph: Automatic Code Review via Simplified AST Driven Graph Convolutional Network, Knowledge-Based Systems, 2022, 252, 109452, doi: 10.1016/j.knosys.2022.109450.

[51] DUAN X., WU J., DU M., LUO T., YANG M., WU Y. MultiCode: A Unified Code Analysis Framework based on Multi-type and Multi-granularity Semantic Learning. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2021, pp. 359–364.

[52] WALUNJ V., GHARIBI G., ALANAZI R., LEE Y. Defect Prediction Using Deep Learning with Network Portrait Divergence for Software Evolution. Empir Software Eng., 2022, 27, 118.

[53] TOSUN A., BENER A.B., AKBARINASAJI S. A systematic literature review on the applications of Bayesian networks to predict software quality, Software Quality Journal, 2017, 25, pp. 273–305.

[54] ZHENG J. Cost-sensitive boosting neural networks for software defect prediction, Expert Systems with Applications, 2010, 37, pp. 4537–4543.

[55] WANG T., LI W., SHI H., LIU Z. Software defect prediction based on classifiers ensemble, Journal of Information & Computational Science, 2011, 8, pp. 4241–4254.

[56] TWALA B. Predicting software faults in large space systems using machine learning techniques, Defence Science Journal, 2011, 61, p. 306.

[57] SIERS M.J., ISLAM M.Z. Cost sensitive decision forest and voting for software defect prediction. In: Pacific Rim International Conference on Artificial Intelligence, 2014, pp. 929–936.