*Computers, Materials & Continua* CMC, vol.64, no.1, pp.471-499, 2020

# KAEA: A Novel Three-Stage Ensemble Model for Software Defect Prediction

**Nana Zhang[1], Kun Zhu[1], Shi Ying[1,*] and Xu Wang[2]**

**Abstract:** Software defect prediction is a research hotspot in the field of software engineering. However, due to the limitations of current machine learning algorithms, we can't achieve good effect for defect prediction by only using machine learning algorithms. In previous studies, some researchers used extreme learning machine (ELM) to conduct defect prediction. However, the initial weights and biases of the ELM are determined randomly, which reduces the prediction performance of ELM. Motivated by the idea of search based software engineering, we propose a novel software defect prediction model named KAEA based on kernel principal component analysis (KPCA), adaptive genetic algorithm, extreme learning machine and Adaboost algorithm, which has three main advantages: (1) KPCA can extract optimal representative features by leveraging a nonlinear mapping function; (2) We leverage adaptive genetic algorithm to optimize the initial weights and biases of ELM, so as to improve the generalization ability and prediction capacity of ELM; (3) We use the Adaboost algorithm to integrate multiple ELM basic predictors optimized by adaptive genetic algorithm into a strong predictor, which can further improve the effect of defect prediction. To effectively evaluate the performance of KAEA, we use eleven datasets from large open source projects, and compare the KAEA with four machine learning basic classifiers, ELM and its three variants. The experimental results show that KAEA is superior to these baseline models in most cases.

**Keywords:** Software defect prediction, KPCA, adaptive genetic algorithm, extreme learning machine, Adaboost.

## 1 Introduction

As the software scale and complexity increase, the number of generated defects also increases dramatically, which has a major impact on software maintenance [Liu, Musial and Chen (2011)]. Therefore, how to quickly and accurately predict the defects hidden in the software in advance has become a tough challenge for software developers. To solve this problem, some researchers propose software defect prediction technology, which is to

[1] School of Computer Science, Wuhan University, Wuhan, 430072, China.

[2] Department of Computer Science, Vrije University Amsterdam, Amsterdam, 1081HV, The Netherlands.

[*] Corresponding Author: Shi Ying. Email: yingshl@whu.edu.cn.

Received: 11 February 2020; Accepted: 02 April 2020.

mine and extract historical information and code information in the process of software development, and establish a specific prediction model by mathematical statistics, machine learning algorithms and other methods to predict the software defects. Software defect prediction technology can not only help developers determine the priority of software testing and debugging, but also recommend software components that may be defective.

Previous studies mainly used the following two methods to conduct software defect prediction: (1) Combining traditional features or manually designing new features. Traditional features can be divided into two categories: static code features and process features. Static code features, such as Halstead scientific metrics [Maurice (1977)] and Mc-Cabe loop complexity [McCabe (1976)]; process features, such as developer personal features [Jiang, Tan and Kim (2013); Ostrand, Weyuker and Bell (2010)] and cooperation between developers [Lee, Nam, Han et al. (2011); Menzies, Milton, Turhan et al. (2010); Pinzger, Nagappan and Murphy (2008); Weyuker, Ostrand and Bell (2007)]. (2) Machine learning-based methods, including various tasks of data processing and modeling. Many machine learning algorithms are designed to support these tasks, and each algorithm has its own data requirement and different complexity. For example, Lessmann et al. [Lessmann, Baesens, Mues et al. (2008)] systematically compare 22 different machine learning methods, which are mainly divided into six categories: statistical method, k-nearest neighbor, neural network, support vector machine, decision tree and integration algorithm. In this paper, we propose a novel method that combines intelligent algorithm (adaptive genetic algorithm) and machine learning algorithms (KPCA, extreme learning machine, Adaboost) to conduct software defect prediction.

Since irrelevant and redundant features in the software defect may degrade the performance of the prediction model, we need to extract optimal features to reveal the intrinsic structure of the defect data, which is crucial to construct effective defect prediction model. In this paper, we leverage KPCA [Schölkopf, Smola and Müller (1997)], a non-linear extension of PCA, to project the original data into a latent high-dimensional feature space in which the mapped features can properly characterize the complex defect data structures and increase the probability of linear separability of the defect data. The literature [Xu, Liu, Luo et al. (2019)] has proved that KPCA is a very advanced feature extraction method.

The ELM randomly generates the connection weights between the input layer and the hidden layer and the biases of the hidden layer neurons, and does not need to be adjusted during the training process. Moreover, the ELM only needs to set the number of neurons in the hidden layer to obtain the unique optimal solution. Compared with traditional training methods, the ELM has the advantages of fast learning rate and good generalization performance. However, the selection of initial connection weights and biases has a great impact on network prediction error, but we cannot obtain them accurately. Therefore, for this shortcoming of ELM, we leverage adaptive genetic algorithm to determine the optimal initial weights and biases of ELM in order to minimize prediction error. Firstly, the individual of adaptive genetic algorithm represents the initial weights and biases of ELM, and the test error of ELM is used as the fitness function to calculate the fitness value of the individual. Then we can find the individual with the minimum error through the selection, crossover and mutation operations, which

is the optimal initial weights and biases of the extreme learning machine.

In addition, we also use the Adaboost algorithm to integrate multiple ELM basic predictors optimized by adaptive genetic algorithm into a strong predictor, which can further improve the effect of defect prediction.

Based on the above analysis, we propose a novel software defect prediction model named KAEA (KPCA-Adaptive Genetic Algorithm-Extreme Learning Machine-Adaboost) based on KPCA, adaptive genetic algorithm, extreme learning machine and Adaboost algorithm in this paper.

The main contributions of this paper are as follows:

(1) In this paper, we propose a novel defect prediction model named KAEA, which leverages an advanced feature extraction method-KPCA to extract optimal features that revealed the intrinsic structure of the defect data.

(2) We adopt adaptive genetic algorithm to optimize initial weights and biases of extreme learning machine, thereby boosting the generalization capacity and and prediction performance of ELM.

(3) We also use Adaboost algorithm to integrate multiple ELM basic predictors optimized by the adaptive genetic algorithm into a strong predictor.

(4) We conduct a large number of defect prediction experiments on eleven datasets from large open source projects, and compare the KAEA with four machine learning basic classifiers, ELM and its three variants. The experimental results verify that our KAEA model can achieve better results than baseline models in most cases.

The rest of this paper is organized as follows. Section 2 describes the background and related work. Section 3 introduces data preprocessing. Section 4 details our proposed KAEA model. Section 5 shows experimental setup, including datasets, evaluation metrics, baseline models and parameter setup. Section 6 evaluates the performance of our KAEA model. Section 7 provides a discussion about the proposed method. Section 8 describes the threats to our work. We conclude this paper and describe future work in Section 9.

## 2 Background & related work

### *2.1 Extreme learning machine*

Extreme learning machine is first proposed by Huang et al. [Huang, Zhu and Siew (2006); Huang, Chen and Siew (2006)], which is a single hidden-layer feedforward neural network (SLFN). The initial weights and hidden layer biases are randomly assigned at first. The basic network structure of ELM is as shown in Fig. 1. ELM is a three-layer neural network that contains input layer, hidden layer and output layer.

Given N training samples $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i = [x_{i1}, x_{i2}, \dots, x_{im}]^T \in R^m$ ($m$ is the number of the input neurons which is equal to the number of the input features) and $y_i = [y_{i1}, y_{i2}, \dots, y_{in}]^T \in R^n$ ($n$ is the number of the output neurons which is equal to the number of classes). The input weight matrix is represented by $w_j = [w_{j1}, w_{j2}, \dots, w_{jm}]^T$, $\theta_j = [\theta_{j1}, \theta_{j2}, \dots, \theta_{jn}]^T$ is the weight vector connecting the $i$th hidden node with output neurons, $b_j$ is the bias of the $j$th hidden neuron. ELM can be mathematically formulated as follows:

$$\sum_{j=1}^{\tilde{N}} \theta_j g\left(w_j \cdot x_i + b_j\right) = o_i, i = 1,2,\ldots,N \ , \tag{1}$$

where $\tilde{N}$ denotes the number of hidden nodes, *g(.)* denotes the activation function, $o_i$ denotes the final output label of the instance $x_i$.

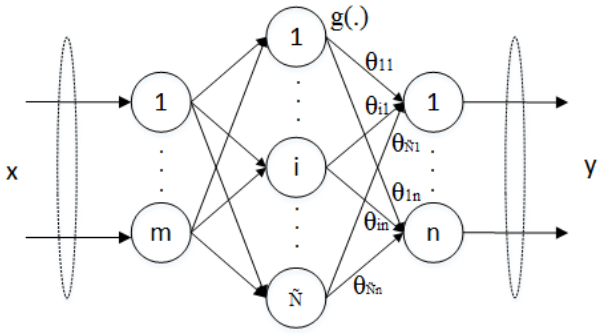


**Figure 1:** The basic network structure of ELM

The Eq. (1) can be written compactly as:

$$\mathrm{G}\theta = \mathrm{O}, \tag{2}$$

where the output of the hidden layer for all the training instances is represented by *G*, which is shown in the following Eq. (3):

$$\mathrm{G} = \begin{bmatrix} g(w_1 \cdot x_1 + b_1) & \cdots & g(w_{\tilde{N}} \cdot x_1 + b_{\tilde{N}}) \\ \vdots & \ddots & \vdots \\ g(w_1 \cdot x_N + b_1) & \cdots & g(w_{\tilde{N}} \cdot x_N + b_{\tilde{N}}) \end{bmatrix}_{N \times \tilde{N}} \tag{3}$$

$\theta$ is shown in the following Eq. (4):

$$\theta = \begin{bmatrix} \theta_{11} & \cdots & \theta_{1n} \\ \vdots & \ddots & \vdots \\ \theta_{\tilde{N}1} & \cdots & \theta_{\tilde{N}n} \end{bmatrix}_{\tilde{N} \times n} \tag{4}$$

*O* is shown in the following Eq. (5):

$$\mathrm{O} = \begin{bmatrix} o_{11} & \cdots & o_{1n} \\ \vdots & \ddots & \vdots \\ o_{N1} & \cdots & o_{Nn} \end{bmatrix}_{N \times n} \tag{5}$$

The standard SLFNs can approximate these *N* instances with zero error. The error of ELM is $\sum_{i=1}^{N} ||o_i - y_i|| = 0$ and there exist $\theta_j$, $w_j$ and $b_j$ such that $\sum_{j=1}^{\tilde{N}} \theta_j g\left(w_j \cdot x_i + b_j\right) = y_i$.

Then, we need to solve the following formula:

$$\mathrm{G}\theta = \mathrm{Y}, \tag{6}$$

where *Y* denotes the target output matrix.

It is clear that only $\theta$ is unknown in Eq. (6), and we can adopt the least square algorithm to acquire its solution, which could be described as follows:

$$\hat{\theta} = \mathrm{G}^{+}\mathrm{Y}, \tag{7}$$

where $G^+$ denotes the Moore-Penrose generalized inverse of the hidden layer output matrix $G$, which can guarantee that the solution is the least-norm least-square solution of Eq. (6).

Finally, we get the classification function of ELM, as shown in Eq. (8):

$$f(x) = g(x)\hat{\theta} = g(x)G^{+}Y. \quad (8)$$

Some researchers have applied ELM to process various tasks. Zhang et al. [Zhang and Zhang (2015)] propose a domain adaptation ELM to solve the sensor drift problem in the E-nose system. Moreover, the transfer learning ELM has also been proposed, which regularizes the difference of the source parameters and target parameters [Chen, Jiang and Jin (2018)]. According to M-estimation theory, Kai et al. [Kai, Qi, Yao et al. (2016)] further propose a unified robust ELM framework, and leverage $l_1$-norm and $l_2$-norm regularized terms respectively to avoid overfitting, as well as four kinds of loss functions to improve noise insensitivity of ELM. Liu et al. [Liu, Zhang, Deng et al. (2017)] propose a unified subspace transfer framework based on ELM, which learns a subspace that jointly minimizes the mean distribution discrepancy (MMD) and maximums margin criterion (MMC). Zhang et al. [Zhang and Zhang (2016)] propose an ELM-based domain adaptation (EDA) for visual knowledge transfer and extend the EDA to multi-view learning.

### *2.2 Feature selection and extraction*

Recently, feature extraction or selection techniques have been introduced into the field of software defect prediction, which can solve high dimensionality problem of the software defect dataset by eliminating irrelevant and redundant features [Kondo, Bezemer, Kamei et al. (2019); Xu, Liu, Yang et al. (2016)]. Feature selection techniques decrease the number of features in a model by selecting the most important, representative features, while feature extraction techniques reduce the number of features by creating new, combined features from the original defect features.

Most previous studies use feature selection techniques to conduct software defect prediction [Menzies, Greenwald and Frank (2007); Ghotra, Mcintosh, Hassan et al. (2017)]. Menzies et al. leverage the information gain technique to rank features based on their importance in a defect prediction model. The experimental result shows that the performance of the prediction model only depends on a small set of features. Xu et al. [Xu, Liu, Yang et al. (2016)] investigate the impact of feature selection techniques on the performance of the prediction model. The experimental results show that the prediction performance of different feature selection techniques varies significantly on all the datasets. Ghotra et al. [Ghotra, Mcintosh, Hassan et al. (2017)] also summary the impact of feature selection techniques for defect prediction. They observe that correlation-based feature selection outperforms other feature selection techniques. Yu et al. [Yu, Ma, Ma et al. (2017)] propose a feature selection method based on Feature Spectral Clustering and feature Ranking (FSCR) to predict the number of software defects.

Feature extraction techniques have not been studied as extensively for software defect prediction. Most researchers use principal component analysis (PCA) to conduct software defect prediction [D'Ambros, Lanza and Robbes (2010); Nagappan, Ball and Zeller (2006)]. For instance, D'Ambros et al. [D'Ambros, Lanza and Robbes (2010)] utilize

PCA to avoid the problem of multicollinearity [Farrar and Glauber (1967)] among the independent variables in software defect prediction.

Different from previous studies, we leverage an advanced feature extraction method-kernel principal component analysis (KPCA) to extract optimal features that revealed the intrinsic structure of the defect data in this paper.

### *2.3 Software defect prediction*

Software defect prediction is a research hotspot in the field of software engineering. Previous researchers have proposed some software defect prediction technologies [Jiang, Tan and Kim (2013); Lee, Nam, Han et al. (2011); Rahman and Devanbu (2013)], which are used to discover and predict software defects in advance, so as to improve software product quality and allocate test resources reasonably.

Some researchers extract features from the software history repository to construct defect prediction classifier based on machine learning algorithm, including bayesian belief networks [Amasaki, Takagi, Mizuno et al. (2003)], support vector machine (SVM) [Elish and Elish (2008)], decision tree (DT) [Khoshgoftaar and Seliya (2002); Wang, Shen and Chen (2012)], neural network (NN) [Paikari, Richter and Ruhe (2012); Thwin and Quah (2005)]. Wang et al. [Wang, Shen and Chen (2012)] and Khoshgoftaar et al. [Khoshgoftaar and Seliya (2002)] study tree-based machine learning algorithm. The experimental results show that tree-based algorithm can improve the performance of defect prediction. Lu et al. [Lu, Cukic and Culp (2014); Lu, Kocaguneli and Cukic (2014)] construct a defect prediction model based on active learning, and further improve the performance of the defect prediction model by means of dimension reduction and feature subset selection. Lessmann et al. [Lessmann, Baesens, Mues et al. (2008)] compare 22 kinds of supervised learning algorithms and find that the optimal learning algorithms are different for different projects, and the performance gap between the optimal 17 machine learning algorithms is not significant. Tantithamthavorn et al. [Tantithamthavorn, McIntosh, Hassan et al. (2016)] further analyze the effect of parameter values for different machine learning methods. They believe that the setup of optimal parameter values would have a great impact on the performance of the final defect prediction model. Guo et al. [Guo, Ma, Cukic et al. (2004)] apply random forest to defect prediction and find that its prediction performance is superior to that of traditional statistical-based algorithms. Yang et al. [Yang, Yin, Xu et al. (2008)] use AP (Affinity Propagation) clustering algorithm to predict the files that may contain defects, and achieve good prediction results. Li et al. [Li, Zhang, Wu et al. (2012)] propose a two-stage semi-supervised ensemble learning software defect prediction method. The results show that the method has better prediction capacity for unbalanced data compare with the classical machine learning methods.

Some researchers apply deep learning to software defect prediction. Yang et al. [Yang, Lo, Xia et al. (2015)] combine 14 change-level features through a deep belief network (DBN) to generate new features for change-level defect prediction. Wang et al. [Wang, Liu and Tan (2016)] propose the DBN-CP method, which shows that the semantic features generated based on representation learning could better capture the common defect features and apply these features to cross-project defect prediction. Wang et al. [Wang, Zhang, Jing et al. (2016)] propose a SemiBoost software defect prediction method-NSSB based on non-

negative sparse graphs, and use Adaboost algorithm to improve the performance of the model. The experimental results show that the NSSB method can effectively solve the class imbalance problem and label instances insufficiency problem.

## 3 Data preprocessing

For software defect datasets, data preprocessing mainly includes two parts: data imbalance processing and data normalization.

### *3.1 Data imbalance processing*

The defect dataset is usually unbalanced, and the number of defective instances is far less than the number of non-defective instances. As can be seen from Tab. 1, the defect rate of all datasets is less than 30%, which results in the relatively low prediction effect for a few classes. Therefore, in order to construct the optimal defect prediction model, we first need to conduct data imbalance processing for these datasets.

The SMOTE (Synthetic Minority Over-sampling Technique) algorithm [Chawla, Bowyer, Hall et al. (2002)] has been widely used in the processing of unbalanced data as a classical oversampling method, and has achieved satisfactory results. In this paper, we adopt the SMOTE algorithm to conduct data imbalance processing. The algorithm synthesizes a few classes through certain strategies, and increases the number of instances of a few classes. In addition, the algorithm is an improved scheme based on random oversampling algorithm, which can effectively solve the over-fitting phenomenon caused by simply replicating instances of a few classes.

This step is critical to the performance of software defect prediction, because it helps the trained predictor not bias toward non-defective modules (majority classes) [Zeng, Xiao, Wang et al. (2019)], thereby improving the performance of defect prediction.

### *3.2 Data normalization*

The distribution of the feature values on defect dataset is of large difference, even not in the same order of magnitude. If the original feature values are directly used for defect prediction, the role of higher numerical values in the comprehensive analysis will be highlighted, and relatively reduce the effect of lower numerical values. Therefore, in order to ensure the reliability of the prediction results, we need to normalize the original defect dataset.

In this paper, we use the mapminmax normalization function, the call format of the function is as shown in Eq. (9):

$$[\mathrm{Y}, \mathrm{PS}] = \mathrm{mapminmax}(\mathrm{X}, \mathrm{YMIN}, \mathrm{YMAX}), \tag{9}$$

where $X$ is the original defect dataset; $YMIN$ is the minimum value of each row on the array, and the default value is -1; $YMAX$ is the maximum value of each row on the array, and the default value is 1; $Y$ is the normalized data; $PS$ is a structure after data normalization, including maximum, minimum, and average values, which can be used to the normalization of the test data.

## 4 Methodolody

In this paper, we propose a software defect prediction model named KAEA, which consists of three stages: (1) Feature extraction based on KPCA. The KAEA model leverages KPCA to extract optimal features, which can reveal the intrinsic structure of the defect data. (2) The initial weights and biases of ELM optimized by adaptive genetic algorithm. The model uses adaptive genetic algorithm to optimize initial weights and biases of ELM, so as to find the initial weights and biases that can minimize the prediction error, that is, the optimal initial weights and biases. (3) Software defect prediction based on the KAEA model. We use Adaboost algorithm to integrate multiple ELM basic predictors optimized by the adaptive genetic algorithm into a strong predictor, which makes the integrated strong predictor better for software defect prediction. The framework of the KAEA model is shown in Fig. 2.

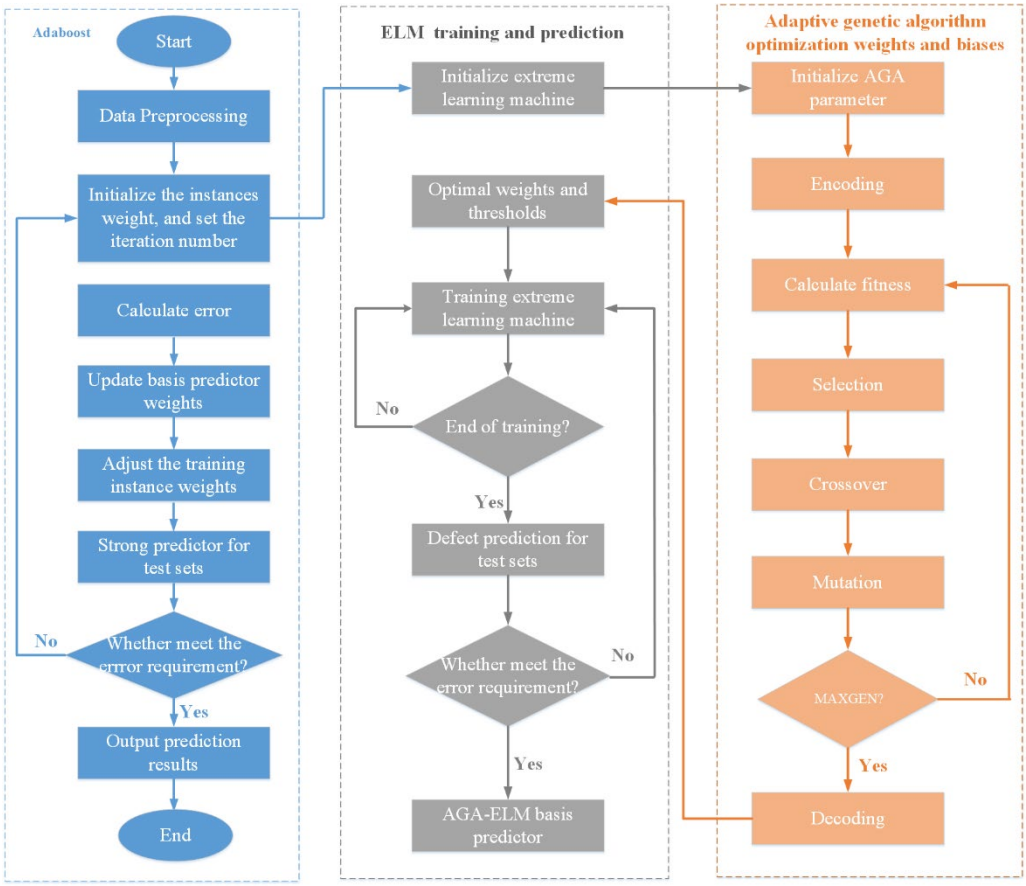


**Figure 2:** The framework of the KAEA model

### *4.1 Feature extraction based on KPCA*

In the first stage, we conduct feature extraction with KPCA, which can extract representative features and characterize the complex defect data structures. KPCA can

project the original data point within a low-dimensional feature space into a new point within a high-dimensional feature space by leveraging a nonlinear mapping function.

Assuming that $x$ is mapped into $u$ through the corresponding function $\rho$, its definition is as follows:

$$\mathrm{u} = \rho(\mathrm{x}) \tag{10}$$

The kernel function maps data points to the corresponding feature space, and the data in the mapped feature space satisfies the concentration condition, i.e.,

$$\sum_{i=1}^{N} \rho(x_i) = 0 \tag{11}$$

The covariance matrix $C$ in the feature space is as follows:

$$\mathrm{C} = \frac{1}{N}\sum_{i=1}^{N} \rho(x_i)\rho(x_i)^T. \tag{12}$$

Finding the eigenvalues $\lambda$ and eigenvectors $w$ of the covariance matrix $C$ through the following equation:

$$\mathrm{Cw} = \lambda \mathrm{w}. \tag{13}$$

We multiply both sides of Eq. (13) by $\rho(x_t)^T$, i.e.,

$$\rho(x_t)^T \mathrm{Cw} = \lambda \rho(x_t)^T w. \tag{14}$$

The coefficient $\xi_j$ can linearly express the eigenvector $w$ with $\rho(x_j)$, i.e.,

$$\mathrm{w} = \sum_{j=1}^{N} \xi_j \rho(x_j). \tag{15}$$

Then, Eq. (14) can be rewritten by substituting Eqs. (12) and (15) as following equation:

$$\frac{1}{N}\rho(x_t)^T \sum_{i=1}^{N} \rho(x_i)\rho(x_i)^T \sum_{j=1}^{N} \xi_j \rho\left(x_j\right) = \lambda \rho(x_t)^T \sum_{j=1}^{N} \xi_j \rho(x_j). \tag{16}$$

The kernel matrix $M$ can be expressed in the following equation:

$$M_{ij} = \mathrm{m}\left(x_i, x_j\right) = \rho(x_i)^T \rho\left(x_j\right). \tag{17}$$

Then, Eq. (16) can be rewritten by substituting Eq. (17) as following equation:

$$\frac{1}{N}\sum_{t=1,i=1}^{N} M_{t,i} \sum_{i=1,j=1}^{N} \xi_j M_{i,j} = \lambda \sum_{t=1,j=1}^{N} \xi_j M_{t,j}\,. \tag{18}$$

The Eq. (18) is rewritten as:

$$M^2 \xi = \lambda \xi N M. \tag{19}$$

That is:

$$\mathrm{M}\xi = \lambda \xi \mathrm{N}. \tag{20}$$

Before applying KPCA, assuming that the mapped data points are centralized. If not, we need to use the Gram matrix $\bar{M}$ to replace the kernel matrix M, so as to complete the mean centralizing. Its equation is as follows:

$$\bar{M} = O_N M - M O_N + O_N M O_N, \tag{21}$$

where $O_N$ represents the n×n matrix that all values equal to 1/n.

Then, we only need to adjust the following equation:

$$\bar{M}\xi = \lambda \xi N. \tag{22}$$

After obtaining eigenvalues and eigenvectors, the projection of the test instance in the feature vector space can be expressed as follows:

$$w^m \cdot \rho(x) = \sum_{i=1}^{N} \xi_i \rho(x_i)\rho(x). \tag{23}$$

After replacing the inner product with a kernel function, the Eq. (23) is rewritten as:

$$w^m \cdot \rho(x) = \sum_{i=1}^{N} \xi_i M(x_i, x). \tag{24}$$

The Radial Basis Function (RBF) kernel can be defined as follows:

$$\mathrm{m}(x_i, x_j) = \exp\left(-\frac{\left||x_i - x_j|\right|^2}{2\sigma^2}\right). \tag{25}$$

After performing feature extraction with KPCA, the original training data are transformed to a new dataset. Next, we will use the new defect dataset after feature extraction for further processing.

### *4.2 The initial weights and biases of ELM optimized by adaptive genetic algorithm*

We use adaptive genetic algorithm to optimize network weights and biases. Firstly, the individual of adaptive genetic algorithm represents the initial weights and biases of ELM, and the test error of the network is used as the fitness function to calculate the fitness value of the individual. Then, we can find the individual with the minimum error through the selection, crossover and mutation operations, which is the optimal initial weights and biases of ELM.

#### *4.2.1 Encoding*

We adopt matrix encoding to encode the individuals, and each individual represents initial weights and biases of the network. More specifically, each individual is composed of two parts: the connection weights between the input layer and the hidden layer, the biases of the hidden layer. We adopt matrix encoding to represent the initial weights and biases of ELM for each individual instead of other encoding strategies, such as vector coding and binary coding. This is because matrix encoding makes the ELM easy to execute decoding, which is suitable for the training process of ELM network. However, for vector encoding, the decoding process is very complex, which may increase the difficulty of network training. For binary encoding, the individuals are encoded as strings of binary bits. But the length of each individual will increase when the network structure becomes more complicated, so the process of encoding and decoding also becomes very complex. Therefore, matrix encoding is more suitable for the training of ELM network. We take a 2-3-2 network structure as an example, as shown in Fig. 3.

$$w_1 = \begin{bmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \\ w_{15} & w_{25} \end{bmatrix}, b_1 = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}, \tag{26}$$

where $w_1$ is the weights from the input layer to the hidden layer, $b_1$ is the biases of the hidden layer.

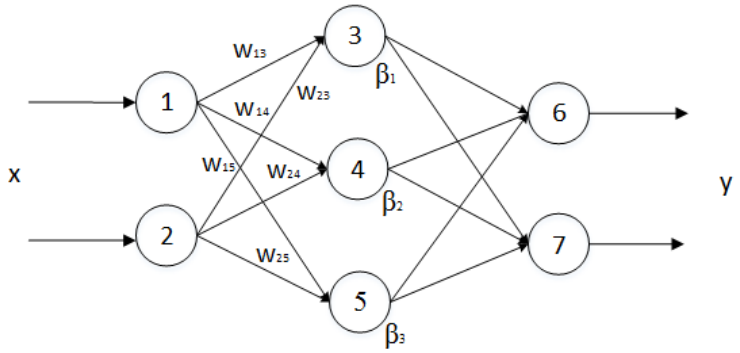


**Figure 3:** ELM with encoded weights and biases

After encoding, the initial weights and biases of ELM are converted into the genetic space with chromosome form, so as to complete the population initialization.

### *4.2.2 Fitness function*

In this paper, when ELM is used for defect prediction, the mean square error between the actual output value and the predicted output value is taken as the output of objection function, and calculate the fitness value of each individual by the mean square error. The smaller the mean square error of the individual, the larger the fitness value and the better the individual. We can find the individual with the minimum error, that is, the optimal initial weights and biases of the network.

For a three-layer ELM, given $N$ training instances $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i = [x_{i1}, x_{i2}, \dots, x_{im}]^T \in R^m$ and $y_i = [y_{i1}, y_{i2}, \dots, y_{in}]^T \in R^n$. According to Eq. (8), we get the output f(x) of the output layer.

So, the equation of the fitness function is as shown in Eq. (28):

$$mse = \frac{1}{N}\sum_{n=1}^{N}(\mathrm{f(x)} - y)^2 , \tag{27}$$

$$fitness = \frac{1}{1+mse}, \tag{28}$$

where *mse* is the mean square error of the network, *f(x)* is the predicted output value of ELM; *y* is the actual output value of ELM.

### *4.2.3 Adaptive crossover rate and mutation rate*

The crossover rate $P_c$ and the mutation rate $P_m$ can adaptively change with the fitness value. When the individual fitness value is the local optimal value, the crossover rate and mutation rate will increase; otherwise, they will decrease.

In this paper, we use Eqs. (29) and (30) to calculate adaptive crossover rate $P_c$ and mutation rate $P_m$, respectively:

$$P_c = \begin{cases} k_1, & \mathrm{f} < F_{avg} \\ k_3, & \mathrm{f} \geq F_{avg} + k_2 \cdot (F_{max} - F_{avg}) \\ k_4 \cdot (F_{max} - f)/(F_{max} - F_{avg}), & \text{else} \end{cases}, \tag{29}$$

where $F_{max}$ is the maximum fitness value of the individual, $F_{avg}$ is the average fitness value of the individual; f is the larger one between the two individuals fitness values; $k_1$, $k_2$, $k_3$ and $k_4$ are constants between (0, 1).

$$P_m = \begin{cases} k_5, & f' < F_{avg} \\ k_7, & f' \geq F_{avg} + k_6 \cdot (F_{max} - F_{avg}), \\ k_8 \cdot (F_{max} - f')/(F_{max} - F_{avg}), & \text{else} \end{cases} \tag{30}$$

where $f'$ is fitness value of the individual variation; $k_5$, $k_6$, $k_7$ and $k_8$ are constants between (0, 1).

### *4.3 Software defect prediction based on the KAEA model*

#### *4.3.1 The integrated strong predictor based on Adaboost algorithm*

We use Adaboost algorithm to further improve the effect of software defect prediction, which can integrate multiple ELM basic predictors optimized by the adaptive genetic algorithm into a strong predictor.

Suppose there are $n$ training instances $T=\{(x_i, y_i)|i=1,2,\ldots,n\}$, $M$ predictor output functions $f_m(x)$, $m$=1, 2, ..., $M$, each predictor is an ELM basic predictor optimized by adaptive genetic algorithm. We obtain a strong predictor integrated by $M$ basic ELM predictors through the Adaboost algorithm. The specific steps are as follows:

Step 1: Initialize the weights of the $n$ training instances, as shown in Eq. (31):

$$D_m(i) = \frac{1}{n}, \tag{31}$$

where $D_m(i)$ represents the weight assigned to the instances ($x_i$, $y_i$) in the $m$th iteration.

Step 2: Calculate the error $E_i$ of the basic predictor. According to the instances distribution $D_m$, $m$=1, 2, ..., $M$, we use the above instances to train the $m$th basic predictor, then use the trained basic predictor to predict the defect instances. The absolute value of the basic predictor error is as follows:

$$E_i = \| f_m(x_i) - y_i \|. \tag{32}$$

Step 3: Calculate and update the weight $w_m(i)$ of the basic predictor. We calculate the weight coefficient $w_m(i)$ of the basic predictor according to $\varepsilon_m$:

$$\varepsilon_m = \sum_{i=1}^{n} D_m(i), i = 1, 2, \ldots, n. \tag{33}$$

$$w_m = \frac{1}{2} \ln \frac{1-\varepsilon_m}{\varepsilon_m}. \tag{34}$$

Step 4: Adjust the training instances weight $D_{m+1}(i)$. Adjust the weight of training instances in the next iteration according to the weight coefficient $w_m$:

$$D_{m+1}(i) = \frac{D_m(i)}{B_m} . \exp\left[-w_m y_i f_m(x_i)\right], \tag{35}$$

where $B_m$ is the normalization factor, the purpose is to make the sum of distribution weights equal to 1 when the weight proportion is constant.

Step 5: Obtain the strong predictor KAEA. After training $M$ rounds, we can obtain the $M$-group basic prediction functions $f_m(x)$, then combine the $M$-group basic prediction functions to obtain a strong prediction function $F(x)$, namely the strong predictor KAEA.

$$\mathrm{F(x)} = \sum_{m=1}^{M} w_m \times f_m(x). \quad (36)$$

*4.3.2 Software defect prediction*

We use $M$ ELMs with the optimal initial weights and biases to train the defect instances, so as to form $M$ basic predictors. The Adaboost algorithm is used to integrate the $M$ basic predictors into a strong predictor, so the test set is classified as defective or non-defective.

In this paper, we use 10 times 10-fold cross-validation to evaluate the performance of the KAEA model.

The pseudo code of the KAEA is shown in the algorithm 1:

**Algorithm 1** KAEA

**Input:**

population size: $P_{size}$; the maximum evolution generation of adaptive genetic algorithm: *MAXGEN*; the constants of the crossover rate $P_c$: $k_1$, $k_2$, $k_3$, $k_4$, the constants of the mutation rate $P_m$: $k_5$, $k_6$, $k_7$, $k_8$.

**Output:**

software defect prediction result: $R$

# **The first stage:** Feature extraction based on KPCA

1: Leverage KPCA to extract optimal representative features;

# **The second stage:** The initial weights and biases of ELM optimized by adaptive genetic algorithm

2: Initialize the parameters of the adaptive genetic algorithm: $P_{size}$, MAXGEN, the constants of $P_c$: $k_1$, $k_2$, $k_3$, $k_4$; the constants of $P_m$: $k_5$, $k_6$, $k_7$, $k_8$;

3: Encode for the initial weights and biases of ELM according to Eq. (26), and obtain the initial population;

4: **for** $m_1$=1 to *MAXGEN* **do**

5: Calculate the fitness value of each individual by the mean square error of ELM according to Eq. (28);

6: Conduct the selection operation by the roulette selection;

7: Conduct the crossover operation with crossover rate $P_c$ by the arithmetic crossover operator;

8: Conduct the mutation operation with mutation rate $P_m$ by the adaptive mutation;

9: **end for**

10: Decode the chromosomes composed of the optimal initial weights and biases in the genetic space;

11: Obtain the optimal initial weights and biases of ELM;

# **The third stage:** Software defect prediction based on the KAEA model

12: Initialize the weights of the $n$ training instances;

13: **for** k=1 to 9 **do** # The number of ELM basic predictors is 9

14: Train the ELM with the optimal initial weights and biases;

15: Calculate the error $E_i$ of the basic predictor according to Eq. (32);

16: Calculate and update the weight $w_m$ of the basic predictors according to Eq. (34);

17: Adjust the training instance weight $D_{m+1}(i)$ according to Eq. (35);
18: **end for**
19: Obtain the strong predictor KAEA;
20: Conduct defect prediction by the KAEA predictor, and obtain the final prediction result $R$;
21: **return** $R$;

## 5 Experimental setup

In this section, we will introduce the experimental setup, including datasets, evaluation metrics, baseline models and parameter setup. We conduct the experiments on a 3.6 GHz i7-4790 CPU machine with 8 GB RAM.

### *5.1 Datasets*

In this paper, we use eleven datasets from the NASA data repository, namely KC2, MC1, CM1, JM1, PC1, PC2, PC3, PC4, PC5, MC2 and MW1. The details of these datasets are shown in Tab. 1, which shows the name of the project, number of metrics, number of instances, number of defective instances, number of non-defective instances, defective ratio and imbalance ratio. It can be seen that the defect ratio of PC2 is the smallest, which is 2.15%, and the defect ratio of PC5 is the largest, which is 27.53% from Tab. 1. The number of defects varies from tens to thousands.

Tab. 2 describes the features of eleven datasets, where the first 20 lines describe the common features of datasets, and the last few lines describe the specific features of each dataset. If the dataset has such a feature, it is denoted by ✓, otherwise, it is blank.

We use 10 times 10-fold cross-validation to evaluate the performance of the KAEA model, so each dataset is randomly divided into 10 folds, where 9 folds are used as the training dataset and the remaining 1 fold is used as the test dataset. To further reduce experimental error, we perform 10 times cross-validation and record the average performance.

### *5.2 Evaluation metrics*

To evaluate the performance of the KAEA model, we use three evaluation metrics, namely precision, recall and F1, which have been widely used to evaluate the performance of defect prediction techniques [Fu and Menzies (2017); Xia, Lo, Wang et al. (2016); Liu, Zhou, Yang et al. (2017)], and they can be calculated from the confusion matrix of the classification results. The confusion matrix is shown in Tab. 3. The confusion matrix is used to store the right and wrong decisions made by the prediction model. These instances are divided into true positive (TP), false positive (FP), true negative (FN) and false negative (TN) according to the combination of its actual class and predicted class.

**precision:** The ratio of correctly predicted defect files to all files predicted to be defective, as shown in Eq. (37):

$$precision = \frac{TP}{TP+FP} \tag{37}$$

**Table 1:** The statistics of the NASA datasets

| Project | # of metrics | # of instances | # of defective instances | # of non-defective instances | Defective ratio (%) | Imbalance ratio (%) |
|---|---|---|---|---|---|---|
| **KC2** | 21 | 522 | 107 | 415 | 20.50 | 3.88 |
| **MC1** | 38 | 1988 | 46 | 1942 | 2.31 | 42.22 |
| **CM1** | 37 | 327 | 42 | 285 | 12.84 | 6.79 |
| **JM1** | 21 | 7782 | 1672 | 6110 | 21.49 | 3.65 |
| **PC1** | 37 | 705 | 61 | 644 | 8.65 | 10.56 |
| **PC2** | 36 | 745 | 16 | 729 | 2.15 | 45.56 |
| **PC3** | 37 | 1077 | 134 | 943 | 12.44 | 7.04 |
| **PC4** | 37 | 1458 | 178 | 1280 | 12.21 | 7.19 |
| **PC5** | 38 | 1711 | 471 | 1240 | 27.53 | 2.63 |
| **MC2** | 39 | 125 | 16 | 109 | 12.80 | 6.81 |
| **MW1** | 37 | 253 | 27 | 226 | 10.67 | 8.37 |

**Table 2:** The feature description of eleven datasets

| Features | KC2 | MC1 | CM1 | JM1 | PC1 | PC2 | PC3 | PC4 | PC5 | MC2 | MW1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BRANCH_COUNT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| LOC_CODE_AND_COMMENT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| LOC_COMMENTS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CYCLOMATIC_COMPLEXITY | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| DESIGN_COMPLEXITY | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ESSENTIAL_COMPLEXITY | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| LOC_EXECUTABLE | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_CONTENT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_DIFFICULTY | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_EFFORT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_ERROR_EST | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_LENGTH | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_LEVEL | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_PROG_TIME | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HALSTEAD_VOLUME | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NUM_OPERANDS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NUM_OPERATORS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NUM_UNIQUE_OPERANDS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NUM_UNIQUE_OPERATORS | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| LOC_TOTAL | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| DECISION_DENSITY | | | ✓ | | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| LOC_BLANK | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| CALL_PAIRS | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CYCLOMATIC_DENSITY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| DECISION_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| DESIGN_DENSITY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EDGE_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ESSENTIAL_DENSITY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| PARAMETER_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| GLOBAL_DATA_COMPLEXITY | | ✓ | | | | | | | ✓ | ✓ | |
| GLOBAL_DATA_DENSITY | | ✓ | | | | | | | ✓ | ✓ | |
| MAINTENANCE_SEVERITY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MODIFIED_CONDITION_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NODE_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MULTIPLE_CONDITION_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NORMALIZED_CYLOMATIC_COMPLEXITY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NUMBER_OF_LINES | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| PERCENT_COMMENTS | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CONDITION_COUNT | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

**Table 3:** Confusion matrix

| **Confusion matrix** | | **Predicted** | |
|---|---|---|---|
| | | Positive (P) | Negative (N) |
| **Actual** | True (T) | TP | FN |
| | Flase (F) | FP | TN |

**recall:** The ratio of correctly predicted defect files to all truly defective files, as shown in Eq. (38):

$$recall = \frac{TP}{TP+FN} \tag{38}$$

**F1:** The weighted harmonic mean of precision and recall, as shown in Eq. (39):

$$F1 = \frac{2\times precision\times recall}{precision+recall} \tag{39}$$

### *5.3 Baseline models*

In the paper, we compare the KAEA model with the following eight baseline models: Machine learning basic classifiers: KLR (KPCA-Logistic Regression), KKNN (KPCA-K-Nearest Neighbor), KNB (KPCA-Naive Bayes), KBP (KPCA - Back Propagation).

ELM and its variants: KELM (KPCA-ELM), KGA-ELM (KPCA Genetic Algorithm-ELM), KA-GA-ELM (KPCA Adaboost-Genetic Algorithm-ELM), KAGA-ELM (KPCA Adaptive Genetic Algorithm-ELM).

### *5.4 Parameter setup*

The increase in the number $k$ of basic predictors for Adaboost algorithm can improve the prediction accuracy of the strong predictor. But if $k$ is too large, the time and space costs will be too large, so $k$ is set to 9 in this paper.

The neuron transfer function is the tansig function. The parameters in adaptive genetic algorithm are set as follows: $P_{size}$ =30, $MAXGEN$=100, the values of the constants in $P_c$: $k_1$=0.6, $k_2$=0.7, $k_3$=0.2 and $k_4$=0.3, the values of the constants in $P_m$: $k_5$=0.8, $k_6$=0.6, $k_7$=0.9 and $k_8$=0.3.

## 6 Experimental results

We will introduce the experimental results in the section. We focus on the performance of our proposed KAEA model and answer the following three research questions (RQ):

**RQ1: Does our KAEA model outperform four machine learning basic classifiers for defect prediction?**

To validate the effectiveness of our KAEA model for defect prediction, we need to compare it with the four machine learning basic classifiers, including the LR, KNN, NB, BP models. Since these basic classifiers all use datasets processed by KPCA, they are named KLR, KKNN, KNB and KBP, respectively.

Tabs. 4-6 present the precision, recall and F1 of our KAEA model compared with those of four machine learning basic classifiers respectively for defect prediction. Note that the highest value of each row is marked in bold.

As shown in Tabs. 4-6, the KAEA model performs better than the four baseline models from the point of average precision, recall and F1. Compared with the four baseline models, the KAEA model achieves 86.95%, 88.79% and 87.66% performance on average in terms of precision, recall and F1, respectively. We also find that the performance of KAEA is worse than that of some other models with a small amount of dataset. For example, On the MW1 dataset, the performance of KAEA is worse than that of KNB in terms of precision.

Figs. 4-6 present box-plots of three metrics for KAEA and four machine learning basic classifiers on eleven datasets on average respectively for defect prediction. From Figs. 4-6, we can observe that the maximum value and the median value achieved by the KAEA model are higher than those achieved by other models in terms of precision, recall and F1, respectively, which can fully demonstrate the superiority of our KAEA model, and it is consistent with the observation in Tabs. 4-6. In addition, we can also see that the

performance of the KNB model is significantly lower than that of other models from the point of recall and F1.

Conclusion: Our KAEA model outperforms the four machine learning basic classifiers in terms of precision, recall and F1, especially on the KNB and KBP models. On average, compared with KNB and KBP models, KAEA achieves 46.65%, 24.27% performance improvement in terms of the F1, respectively.

**Table 4:** The precision comparison of KAEA and four basic classifiers

| Dataset | KLR | KKNN | KNB | KBP | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.7812 | 0.8081 | 0.8823 | 0.7832 | **0.8943** |
| **MC1** | 0.7537 | 0.8313 | 0.7264 | 0.6506 | **0.8967** |
| **CM1** | 0.7849 | 0.8029 | 0.7647 | 0.6118 | **0.8563** |
| **JM1** | 0.7087 | 0.7018 | **0.9069** | 0.6450 | 0.7994 |
| **PC1** | 0.8102 | 0.7340 | 0.7676 | 0.7103 | **0.8605** |
| **PC2** | 0.7865 | 0.8151 | 0.6417 | 0.8095 | **0.8511** |
| **PC3** | 0.7898 | 0.7447 | 0.6259 | 0.8240 | **0.8945** |
| **PC4** | 0.8288 | 0.7598 | 0.5643 | 0.7305 | **0.9379** |
| **PC5** | 0.6715 | 0.6437 | 0.5419 | 0.6861 | **0.8889** |
| **MC2** | 0.8000 | 0.7571 | 0.7777 | 0.7976 | **0.8638** |
| **MW1** | 0.6500 | 0.8214 | **0.8723** | 0.7740 | 0.8215 |
| **Average** | 0.7605 | 0.7654 | 0.7338 | 0.7293 | **0.8695** |

**Table 5:** The recall comparison of KAEA and four basic classifiers

| Dataset | KLR | KKNN | KNB | KBP | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.8000 | 0.8236 | 0.1145 | 0.7135 | **0.9113** |
| **MC1** | **0.7972** | 0.7897 | 0.4178 | 0.6512 | 0.7903 |
| **CM1** | 0.8021 | 0.8354 | 0.1428 | 0.7178 | **0.9889** |
| **JM1** | 0.4907 | 0.6197 | 0.0205 | 0.5716 | **0.8528** |
| **PC1** | 0.8102 | 0.8738 | 0.3897 | 0.7085 | **0.9700** |
| **PC2** | 0.8689 | 0.8951 | 0.1877 | 0.4976 | **0.9019** |
| **PC3** | 0.7841 | **0.8036** | 0.5719 | 0.3554 | 0.7926 |
| **PC4** | 0.8970 | 0.8781 | 0.8000 | 0.3659 | **0.9002** |
| **PC5** | 0.6106 | 0.7453 | 0.2240 | 0.6071 | **0.8222** |
| **MC2** | 0.5925 | 0.6571 | 0.2692 | 0.5803 | **0.8700** |
| **MW1** | 0.6093 | 0.6200 | 0.6029 | 0.6100 | **0.9667** |
| **Average** | 0.7330 | 0.7765 | 0.3401 | 0.5799 | **0.8879** |

**Table 6:** The F1 comparison of KAEA and four basic classifiers

| Dataset | KLR | KKNN | KNB | KBP | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.7905 | 0.8158 | 0.2027 | 0.7467 | **0.9027** |
| **MC1** | 0.7748 | 0.8100 | 0.5305 | 0.6509 | **0.8401** |
| **CM1** | 0.7934 | 0.8188 | 0.2407 | 0.6606 | **0.9178** |
| **JM1** | 0.5799 | 0.6582 | 0.0401 | 0.6061 | **0.8252** |
| **PC1** | 0.8102 | 0.7978 | 0.5170 | 0.7094 | **0.9120** |
| **PC2** | 0.8257 | 0.8532 | 0.2905 | 0.6163 | **0.8758** |
| **PC3** | 0.7870 | 0.7730 | 0.5977 | 0.4966 | **0.8405** |
| **PC4** | 0.8615 | 0.8147 | 0.6618 | 0.4876 | **0.9187** |
| **PC5** | 0.6396 | 0.6908 | 0.3169 | 0.6442 | **0.8542** |
| **MC2** | 0.6808 | 0.7036 | 0.4000 | 0.6718 | **0.8669** |
| **MW1** | 0.6290 | 0.7066 | 0.7130 | 0.6823 | **0.8882** |
| **Average** | 0.7429 | 0.7675 | 0.4101 | 0.6339 | **0.8766** |

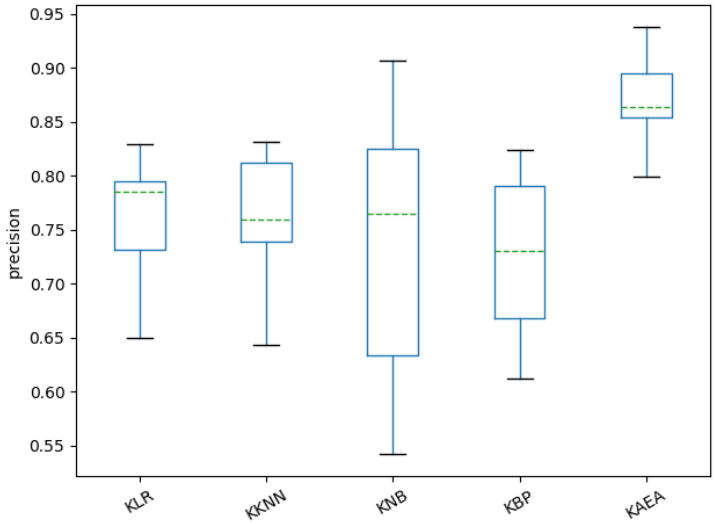


**Figure 4:** The box-plot of precision for KAEA and four basic classifiers on eleven datasets on average

**RQ2: Does our KAEA model outperform ELM and its three variants for defect prediction?**

To further validate the effectiveness of our KAEA model for defect prediction, we also need to compare it with ELM and its three variants, including the KELM, KGA-ELM, KA-GA-ELM and KAGA-ELM, and these models all use datasets processed by KPCA.

Tabs. 7, 8 and 9 present the precision, recall and F1 of our KAEA model compared with those of four baseline models respectively for defect prediction. Note that the highest value of each row is marked in bold.

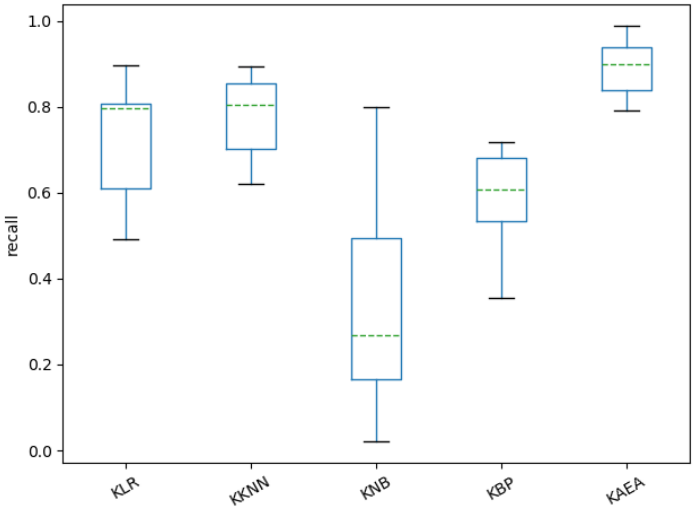


**Figure 5:** The box-plot of recall for KAEA and four basic classifiers on eleven datasets on average

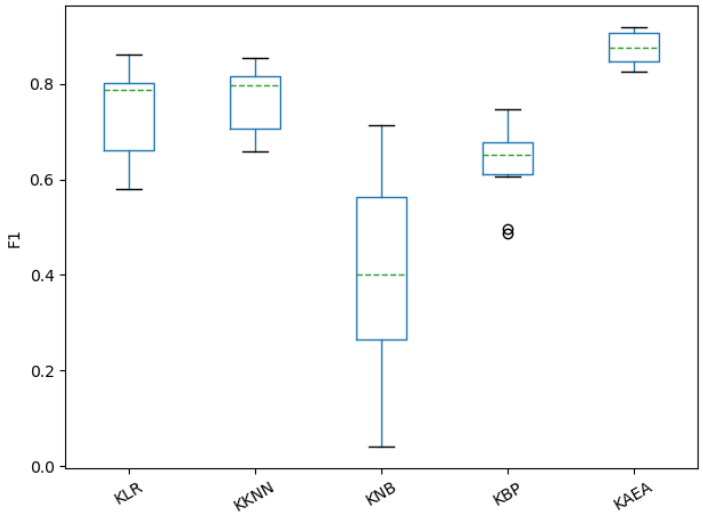


**Figure 6:** The box-plot of F1 for KAEA and four basic classifiers on eleven datasets on average

As shown in Tabs. 7-9, the KAEA model performs better than the four baseline models from the point of average precision, recall and F1. We also find that the performance of the KAEA model is not the best in a few cases. For example, on the PC2 dataset, the performance of KAEA is 85.11%, which is worse than KGA-ELM, KA-GA-ELM and KAGA-ELM in terms of precision. On the MC1, JM1 and PC5 datasets, the performance of our model are 79.03%, 85.28% and 82.22%, which are worse than KAGA-ELM, KA-GA-ELM and KA-GA-ELM model with the best performance respectively in terms of recall.

**Table 7:** The precision comparison of KAEA and four baseline models

| Dataset | KELM | KGA-ELM | KA-GA-ELM | KAGA-ELM | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.8083 | 0.8268 | 0.8316 | 0.8224 | **0.8943** |
| **MC1** | 0.7268 | 0.7839 | 0.8307 | 0.8100 | **0.8967** |
| **CM1** | 0.7816 | 0.8111 | 0.7944 | 0.8163 | **0.8563** |
| **JM1** | 0.7051 | 0.6863 | 0.4966 | 0.5562 | **0.7994** |
| **PC1** | 0.7263 | 0.8040 | 0.8053 | 0.8373 | **0.8605** |
| **PC2** | 0.8378 | 0.9171 | 0.8994 | **0.9430** | 0.8511 |
| **PC3** | 0.8455 | 0.8533 | 0.8364 | 0.8667 | **0.8945** |
| **PC4** | 0.8333 | 0.9059 | 0.7602 | 0.9326 | **0.9379** |
| **PC5** | 0.8308 | 0.8417 | 0.5021 | 0.8398 | **0.8889** |
| **MC2** | 0.8333 | 0.6897 | 0.7407 | 0.7143 | **0.8638** |
| **MW1** | 0.7407 | 0.7308 | 0.7800 | 0.7900 | **0.8215** |
| **Average** | 0.7881 | 0.8046 | 0.7525 | 0.8117 | **0.8695** |

**Table 8:** The recall comparison of KAEA and four baseline models

| Dataset | KELM | KGA-ELM | KA-GA-ELM | KAGA-ELM | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.7823 | 0.8468 | 0.8677 | 0.8065 | **0.9113** |
| **MC1** | 0.7756 | 0.8269 | 0.7544 | **0.8316** | 0.7903 |
| **CM1** | 0.7556 | 0.8111 | 0.9444 | 0.8226 | **0.9889** |
| **JM1** | 0.4574 | 0.5135 | **0.9962** | 0.7922 | 0.8528 |
| **PC1** | 0.6900 | 0.8000 | 0.7650 | 0.8600 | **0.9700** |
| **PC2** | 0.4208 | 0.8507 | 0.9000 | 0.8622 | **0.9019** |
| **PC3** | 0.3852 | 0.5815 | 0.6815 | 0.7634 | **0.7926** |
| **PC4** | 0.3274 | 0.5446 | 0.8869 | 0.8929 | **0.9002** |
| **PC5** | 0.3500 | 0.2806 | **0.9861** | 0.7721 | 0.8222 |
| **MC2** | 0.6000 | 0.8000 | 0.8000 | 0.8200 | **0.8700** |
| **MW1** | 0.6667 | 0.9500 | 0.9633 | 0.9617 | **0.9667** |
| **Average** | 0.5646 | 0.7096 | 0.8678 | 0.8350 | **0.8879** |

Figs. 7-9 present box-plots of three metrics for KAEA and ELM and its three variants on eleven datasets on average respectively for defect prediction. From Figs. 7-9, we can observe that the median value achieved by the KAEA model is higher than that achieved by other models in terms of precision, recall and F1, respectively, which also reflect that our KAEA model can achieve a competitive prediction effect. We can also find that the performance of the KELM model is far from ideal from the point of recall and F1, but it performs well in precision. Moreover, there are more outliers in Fig. 7, but those outliers don't affect the overall performance comparison of the various models.

Conclusion: Our KAEA model outperforms the ELM and its three variants in terms of precision, recall and F1. On average, compared with KELM, KGA-ELM, KA-GA-ELM, and KAGA-ELM models, KAEA achieves 23.77%, 14.17%, 8.42% and 5.69% performance improvement in terms of the F1 respectively.

**Table 9:** The F1 comparison of KAEA and four baseline models

| Dataset | KELM | KGA-ELM | KA-GA-ELM | KAGA-ELM | KAEA |
|---|---|---|---|---|---|
| **KC2** | 0.7951 | 0.8367 | 0.8493 | 0.8144 | **0.9027** |
| **MC1** | 0.7504 | 0.8048 | 0.7907 | 0.8207 | **0.8401** |
| **CM1** | 0.7684 | 0.8111 | 0.8629 | 0.8194 | **0.9178** |
| **JM1** | 0.5549 | 0.5874 | 0.6628 | 0.6535 | **0.8252** |
| **PC1** | 0.7077 | 0.8020 | 0.7846 | 0.8485 | **0.9120** |
| **PC2** | 0.5602 | 0.8826 | 0.8997 | **0.9008** | 0.8758 |
| **PC3** | 0.5293 | 0.6916 | 0.7510 | 0.8118 | **0.8405** |
| **PC4** | 0.4701 | 0.6803 | 0.8187 | 0.9123 | **0.9187** |
| **PC5** | 0.4925 | 0.4208 | 0.6654 | 0.8045 | **0.8542** |
| **MC2** | 0.6977 | 0.7407 | 0.7692 | 0.7635 | **0.8669** |
| **MW1** | 0.7018 | 0.8261 | 0.8620 | 0.8674 | **0.8882** |
| **Average** | 0.6389 | 0.7349 | 0.7924 | 0.8197 | **0.8766** |

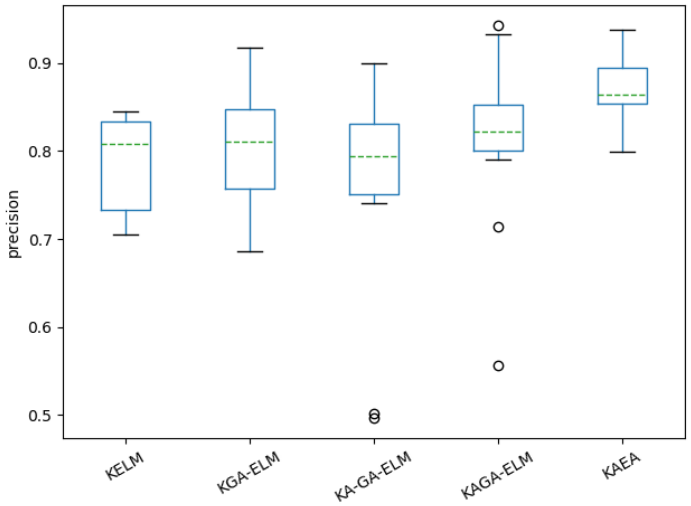


**Figure 7:** The box-plot of precision for KAEA and four baseline models on eleven datasets on average

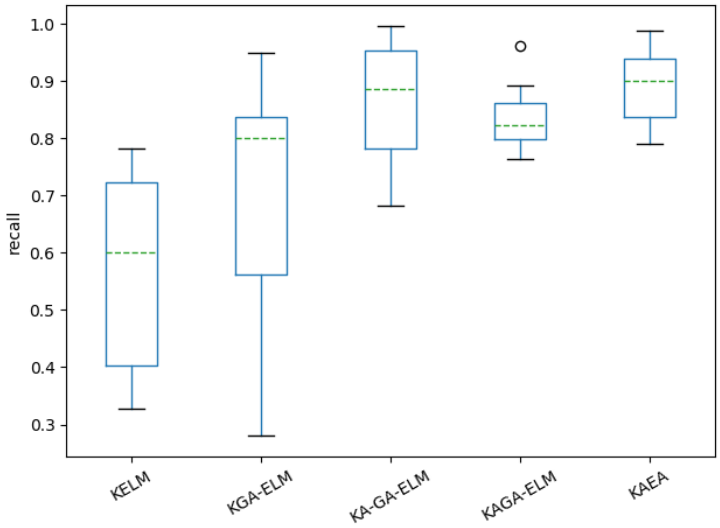


**Figure 8:** The box-plot of recall for KAEA and four baseline models on eleven datasets on average

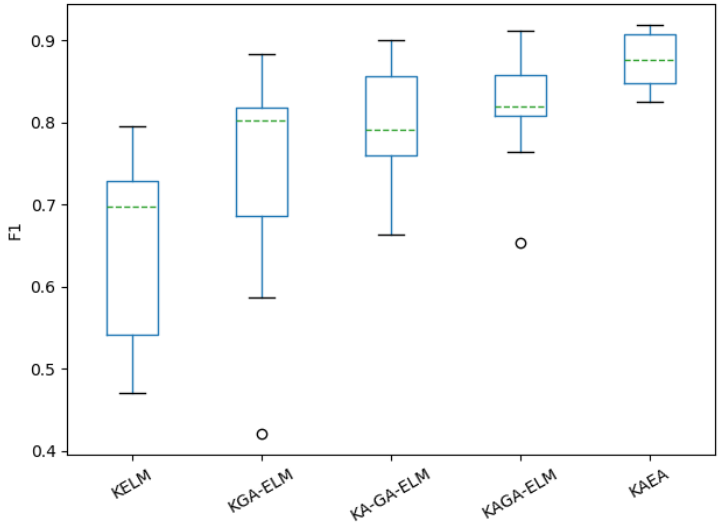


**Figure 9:** The box-plot of F1 for KAEA and four baseline models on eleven datasets on average

**RQ3: Does our KAEA model with the Adaboost ensemble learning method outperform the model without it?**

Fig. 10 presents the precision, recall and F1 of our KAEA model compared with KAGA-ELM (without the Adaboost ensemble learning method) for defect prediction. The evaluation metric values in the comparison figure are the average of the experimental results on the eleven datasets.

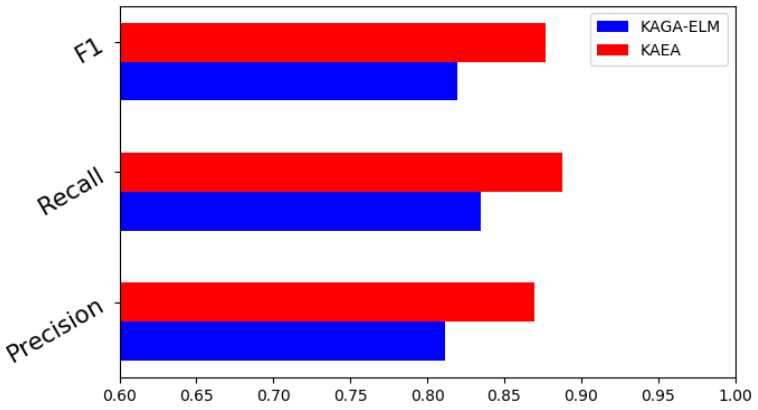


**Figure 10:** The evaluation metrics comparison figure between KAEA and KAGA-ELM

From Fig. 10, we can observe that the performance of KAEA using Adaboost ensemble learning method better than that of KAGA-ELM without the ensemble learning method in terms of precision, recall and F1. More specifically, the average precision (86.95%), recall (88.79%) and F1 (87.66%) by KAEA yields improvement 5.78%, 5.29% and 5.69% compared with KAGA-ELM, respectively. The experimental results verify that the integrated ELM strong predictor can improve the effect of defect prediction.

> Conclusion: Our KAEA model with the Adaboost ensemble learning method outperforms the model without it in terms of precision, recall and F1. KAEA achieves 5.78%, 5.29%, 5.69% performance improvement in terms of the three metrics compared with KAGA-ELM, respectively.

## 7 Discussion

Compared to baseline models in Section 5.3, our KAEA model is the best predictor. This is because the ELM is considered an advanced neural network, which has obvious advantages in classification, including strong classification capacity, good generalization performance. Moreover, we utilize adaptive genetic algorithm (AGA) to optimize initial weights and biases of ELM, and find the optimal initial weights and biases that can minimize the prediction error, thereby further enhance the classification capacity of the ELM. Compared to traditional genetic algorithm (GA), the AGA used in this paper is an improvement on the basic GA. The AGA improves the convergence speed and accuracy of the basic GA by adaptive adjustment of genetic parameters while maintaining the population diversity. We also leverage Adaboost algorithm to integrate multiple ELM basic predictors optimized by the adaptive genetic algorithm into a strong predictor, which makes the integrated strong predictor better for software defect prediction.

## 8 Threat to validity

In this section, we discuss three kinds of validity threats that may have an impact on our experimental results, namely internal validity, external validity and construct validity.

### *8.1 Internal validity*

Internal validity is related to uncontrolled aspects that may affect the experimental results, such as errors in the experiment. We examine our experiment process carefully, but there may still be errors that we do not notice.

### *8.2 External validity*

External validity is related to the quality and universality of the eleven datasets used in the paper, which are often used in previous software defect prediction studies. The scale of these projects is large enough, and these instances are universal enough. In the future, we also plan to further reduce this threat by analyzing more instances of other open source and commercial projects.

### *8.3 Construct validity*

Construct validity involves the applicability of our evaluation metrics. In this paper, we use three evaluation metrics, namely precision, recall and F1, which have been used in previous studies [Fu and Menzies (2017); Xia, Lo, Wang et al. (2016); Liu, Zhou, Yang et al. (2017)], so we believe that construct validity is acceptable.

The experimental design may also affect our experimental results. Recent studies have pointed out that defect prediction models with different parameter setup may produce different results. In order to reduce the threat to the experimental design of parameter setup, we plan to use parameter optimization techniques for more experiments.

## 9 Conclusion and future work

In this paper, we propose a novel defect prediction model named KAEA based on KPCA, adaptive genetic algorithm, ELM and Adaboost algorithm. We first leverage KPCA to extract optimal representative features, so as to reflect the intrinsic structure of the defect data. Then we also use adaptive genetic algorithm to optimize the initial weights and biases of ELM, and obtain the optimal initial weights and biases. Moveover, the Adaboost algorithm is used to integrate multiple ELM basic predictors optimized by adaptive genetic algorithm into a strong predictor, which further improves the effect of defect prediction. We conduct a large number of defect prediction experiments on eleven datasets from large open source projects, and compare the KAEA with four machine learning basic classifiers, ELM and its three variants. The experimental results show that our KAEA model can achieve better results than baseline models in most cases.

In the future, we will evaluate our KAEA model in more open source and commercial projects. In addition, we will use parameter optimization technologies to adjust the parameter setup of the KAEA model, for example, the number of hidden nodes, so as to achieve the optimal defect prediction performance.

**Funding Statement:** This work is supported in part by the National Science Foundation of China (61672392, 61373038), and in part by the National Key Research and Development Program of China (No. 2016YFC1202204).

**Conflicts of Interest:** The authors declare that they have no conflicts of interest to report regarding the present study.

**References**

**Amasaki, S.; Takagi, Y.; Mizuno, O.; Kikuno, T.** (2003): A Bayesian belief network for assessing the likelihood of fault content. *14th International Symposium on Software Reliability Engineering*, pp. 215-226.

**Chawla, N. V.; Bowyer, K. W.; Hall, L. O.; Kegelmeyer, W. P.** (2002): Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, vol. 16, pp. 321-357.

**Chen, C.; Jiang, B.; Jin, X.** (2018): Parameter transfer extreme learning machine basedon projective model. *The International Joint Conference on Neural Networks*, pp. 1-8.

**D'Ambros, M.; Lanza, M.; Robbes, R.** (2010): An extensive comparison of bug prediction approaches. *Proceedings of the 7th International Conference on Mining Software Repositories*, pp. 31-41.

**Elish, K. O.; Elish, M. O.** (2008): Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, vol. 81, no. 5, pp. 649-660.

**Farrar, D. E.; Glauber, R. R.** (1967): Multicollinearity in regression analysis: the problem revisited. *Review of Economics & Statistics*, vol. 49, no. 1, pp. 92-107.

**Fu, W.; Menzies, T.** (2017): Revisiting unsupervised learning for defect prediction. *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pp. 72-83.

**Ghotra, B.; Mcintosh, S.; Hassan, A. E.** (2017): A large-scale study of the impact of feature selection techniques on defect classification models. *Proceedings of the 14th International Conference on Mining Software Repositories*, pp. 146-157.

**Guo, L.; Ma, Y.; Cukic, B.; Singh, H.** (2004): Robust prediction of fault-proneness by random forests. *15th International Symposium on Software Reliability Engineering*, pp. 417-428.

**Huang, G. B.; Chen, L.; Siew, C. K.** (2006): Universal approximation using incremental constructive feedforward networks with random hidden nodes. *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 879-892.

**Huang, G. B.; Zhu, Q. Y.; Siew, C. K.** (2006): Extreme learning machine: theory and applications. *Neurocomputing*, vol. 70, no. 1, pp. 489-501.

**Jiang, T.; Tan, L.; Kim, S.** (2013): Personalized defect prediction. *28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 279-289.

**Kai, C.; Qi, L.; Yao, L.; Yong, D.** (2016): Robust regularized extreme learning machine for regression using iteratively reweighted least squares. *Neurocomputing*, vol. 230, pp. 345-358.

**Kondo, M.; Bezemer, C. P.; Kamei, Y.; Ahmed, E. H.; Osamu, M.** (2019): The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering*, vol. 24, pp. 1925-1963.

**Khoshgoftaar, T. M.; Seliya, N.** (2002): Tree-based software quality estimation models for fault prediction. *Proceedings Eighth Symposium on Software Metrics*, pp. 203-214.

**Lee, T.; Nam, J.; Han, D.; Kim, S.; In, H. P.** (2011): Micro interaction metrics for defect prediction. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 311-321.

**Lessmann, S.; Baesens, B.; Mues, C.; Pietsch, S.** (2008): Benchmarking classification models for software defect prediction: a proposed framework and novel findings. *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485-496.

**Li, M.; Zhang, H.; Wu, R.; Zhou, Z. H.** (2012): Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, vol. 19, no. 2, pp. 201-230.

**Liu, A. G.; Musial, E.; Chen, M. H.** (2011): Progressive reliability forecasting of service-oriented software. *IEEE International Conference on Web Services*, pp. 532-539.

**Liu, J.; Zhou, Y.; Yang, Y.; Lu, H.; Xu, B.** (2017): Code churn: a neglected metric in effort-aware just-in-time defect prediction. *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 11-19.

**Liu, Y.; Zhang, L.; Deng, P.; He, Z.** (2017): Common subspace learning via cross-domainextreme learning machine. *Cognitive Computation*, vol. 9, no. 3, pp. 1-9.

**Lu, H.; Cukic, B.; Culp, M.** (2014): A semi-supervised approach to software defect prediction. *IEEE 38th Annual Computer Software and Applications Conference*, pp. 416-425.

**Lu, H.; Kocaguneli, E.; Cukic, B.** (2014): Defect prediction between software versions with active learning and dimensionality reduction. *IEEE 25th International Symposium on Software Reliability Engineering*, pp. 312-322.

**Maurice, H. H.** (1978): *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

**McCabe, T. J.** (1976): A complexity measure. *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320.

**Menzies, T.; Greenwald, J.; Frank, A.** (2007): Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2-13.

**Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y. et al.** (2010): Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, vol. 17, no. 4, pp. 375-407.

**Nagappan, N.; Ball, T.; Zeller, A.** (2006): Mining metrics to predict component failures. *Proceedings of the 28th International Conference on Software Engineering*, pp. 452-461.

**Ostrand, T. J.; Weyuker, E. J.; Bell, R. M.** (2010): Programmer-based fault prediction. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp. 1-10.

**Paikari, E.; Richter, M. M.; Ruhe, G.** (2012): Defect prediction using case-based reasoning: an attribute weighting technique based upon sensitivity analysis in neural

networks. *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 06, pp. 747-768.

**Pinzger, M.; Nagappan, N.; Murphy, B.** (2008): Can developer-module networks predict failures? *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 2-12.

**Rahman, F.; Devanbu, P.** (2013): How, and why, process metrics are better. *35th International Conference on Software Engineering*, pp. 432-441.

**Schölkopf, B.; Smola, A.; Müller, K. R.** (1997): Kernel principal component analysis. *In International Conference on Artificial Neural Networks*, pp. 583-588.

**Srinivas, M.; Patnaik, L. M.** (1994): Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656-667.

**Tantithamthavorn, C.; McIntosh, S.; Hassan, A. E.; Matsumoto, K.** (2016): Automated parameter optimization of classification techniques for defect prediction models. *IEEE/ACM 38th International Conference on Software Engineering*, pp. 321-332.

**Thwin, M. M. T.; Quah, T. S.** (2005): Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software*, vol. 76, no. 2, pp. 147-156.

**Wang, J.; Shen, B.; Chen, Y.** (2012): Compressed c4. 5 models for software defect prediction. *12th International Conference on Quality Software*, pp. 13-16.

**Wang, S.; Liu, T.; Tan, L.** (2016): Automatically learning semantic features for defect prediction. *IEEE/ACM 38th International Conference on Software Engineering*, pp. 297-308.

**Wang, T.; Zhang, Z.; Jing, X.; Liu, Y.** (2016): Non-negative sparse-based semiboost for software defect prediction. *Software Testing, Verification and Reliability*, vol. 26, no. 7, pp. 498-515.

**Weyuker, E. J.; Ostrand, T. J.; Bell, R. M.** (2007): Using developer information as a factor for fault prediction. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pp. 1-7.

**Xia, X.; Lo, D.; Wang, X.; Yang, X.** (2016): Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1810-1829.

**Xu, Z.; Liu, J.; Luo, X.; Yang, Z.; Zhang, Y. et al.** (2019): Software defect prediction based on kernel PCA and weighted extreme learning machine. *Information and Software Technology*, vol. 106, no. 2, pp. 182-200.

**Xu, Z.; Liu, J.; Yang, Z.; An, G.; Jia, X.** (2016): The impact of feature selection on defect prediction performance: an empirical comparison. *Proceedings of the 27th International Symposium on Software Reliability Engineering*, pp. 309-320.

**Yang, B.; Yin, Q.; Xu, S.; Guo, P.** (2008): Software quality prediction using affinity propagation algorithm. *IEEE International Joint Conference on Neural Networks*, pp. 1891-1896.

**Yang, X.; Lo, D.; Xia, X.; Zhang, Y.; Sun, J.** (2015): Deep learning for just-in-time defect prediction. *IEEE International Conference on Software Quality, Reliability and Security*, pp. 17-26.

**Yu, X.; Ma, Z.; Ma, C.; Gu, Y.; Liu, R. Q. et al.** (2017): FSCR: A feature selection method for software defect prediction. *International Conference on Software Engineering and Knowledge Engineering*, pp. 351-356.

**Zeng, D.; Xiao, Y.; Wang, J.; Dai, Y.; Sangaiah, A. K.** (2019): Distant supervised relation extraction with cost-sensitive loss. *Computers, Materials & Continua*, vol. 60, no. 3, pp. 1251-1261.

**Zhang, L.; Zhang, D.** (2015): Domain adaptation extreme learning machines for drift compensation in e-nose systems. *IEEE Transactions on Instrumentation Measurement*, vol. 64, no. 7, pp. 1790-1801.

**Zhang, L.; Zhang, D.** (2016): Robust visual knowledge transfer via extreme learning machine based domain adaptation. *IEEE Transactions on Image Processing*, vol. 25, no. 10, pp. 4959-4973.