Learning Stretch-Shrink Latent Representations With Autoencoder and K-Means for Software Defect Prediction

VIET ANH PHAN
Department of Information Security, Faculty of Information Technology, Le Quy Don Technical University, Hanoi 100000, Vietnam
e-mail: anhpv@lqdtu.edu.vn

This work was supported in part by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under Grant 102.05-2018.306, and in part by Intelligent Integration Co., Ltd. (INT2), Vietnam, for sponsoring computing resources.

ABSTRACT Detecting defective source code to localize and fix bugs is important to reduce software development efforts. Although deep learning models have made a breakthrough in this field, many issues have not been resolved, such as labeled data shortage and the small size of defective elements. Given two similar programs that differ from each other by an operator or statement, one may be clean while the other may be defective. To address these issues, this study proposes a new deep learning model to facilitate the learning of distinguishing features. The model comprises of three main components: 1) a convolutional neural network-based classifier, 2) an autoencoder, and 3) a k-means cluster. In our model, the autoencoder assists the classifier in generating program latent representations. The k-means cluster provides penalty functions to increase the distinguishability among latent representations. We evaluated the effectiveness of the model according to performance metrics and latent representation quality. The experimental results on the four defect prediction datasets show that the proposed model outperforms the baselines thanks to the generation of sophisticated features.

INDEX TERMS Autoencoder, k-means, convolutional neural networks, latent representations, software defects.

I. INTRODUCTION
Software defect prediction is an active research topic thanks to the great significance in the software industry. According to annual reports, software failures have affected many users and led to serious consequences like financial losses [1], cyber risks [2], and even disasters in autonomous systems [3]. Software companies continuously issue patches to fix bugs through customer feedback or self-investigation. The later the bugs are detected, the greater efforts to fix them are. Thus, predictive models for defective code are efficient tools to enhance software quality and safety.

The traditional methods train the predictive models based on software metrics. The software metrics are statistical measurements of source code such as the complexity, the execution time, the load time, and the instruction path length. Various learning algorithms have been exploited to process the features and build the classifiers including Bayesian networks, Support Vector Machines (SVM), and decision trees [4], [5]. To address the critical issues related to data, several studies focused on feature distillation methods, i.e. autoencoders [6], feature selection [7], and Principal Component Analysis (PCA) [8]. The effects of unbalanced data were investigated [9], and various rebalancing strategies were employed to facilitate model training [10], [11]. Even so, the prediction performance is not high. An unresolved problem is that the existing software metrics reflect insufficient information about syntax and semantics of code [12].

Thanks to automatically learning programs’ features, deep learning has made breakthroughs in the software defect prediction. To build high performance predictors, many networks were designed on representations that contain...
rich information about code syntax and semantics including language tokens [12], token abstract syntax trees (AST) [13], and instruction sequences [14]. These networks encode a whole program into a vector representation. Meanwhile, defects just exist in a tiny part in the program such as an operator, a statement, or a snippet. Our observations on source files have found that similar programs may belong to different groups, and different code structures may both be defective. Consider two similar codes in sub-figures 1a and 1b for finding the greatest common divisor (GCD) of two integers, the defect in the second code is caused by the use of a wrong operator (the division instead of the modulo). In this scenario, the feature sets generated by networks that treat code components with the same role may not be distinguishable.

To pay attention to smaller elements when learning distinguishing features, we propose a new deep learning model including three components with different tasks. The first one is a convolutional classifier for the defective prediction. An autoencoder that encodes the input into latent representations and reconstructs the input from such representations will support the classifier in learning important features. The classifier and the autoencoder tend to keep meaningful information of the whole program. To increase the distinguishability among similar programs with different defective characteristics, a k-means cluster is built on the feature space and provides two penalty functions. The two functions aim to stretch the distance of samples within clusters, and shrink the centroids to reduce the variance.

Figure 1 shows two coding styles of recursive (1a, 1b) and non-recursive (1c, 1d) to compute the GCD of two integers. Since each coding style may be clean or defective, enlarging the distance between samples is necessary for discrimination.

To sum up, the main contributions of the paper are as follows:

- Observing and analyzing carefully the defective characteristics and the difficulty to learn the distinguishing features, even with some deep networks.
- Proposing a new model to not only learn sophisticated features of programs but also enlarge the defective characteristic discrimination ability within each coding style.
- Validating the effectiveness of the proposed method by different perspectives such as performance measures, the ability to deal with the unbalanced data, and the quality of the features.

The rest of the paper is organized as follows. Section II surveys the relevant studies to software defect prediction. Section III describes the proposed model and the training procedure. Section IV presents the baselines and the experimental settings. The results and evaluation are discussed in Section V. We conclude the findings in Section VI.
code, and considering a component whether to be defective or clean depends on specific requirements. Regarding the defect prediction, the survey in [15] revealed a disjunction between research evidence and practitioners’ perceptions.

Initially, handcrafted features namely software metrics were selected to distinguish source code. A software metric is a statistical characteristic of code such as the file size, the complexity, and the execution time. Various common learning and deep learning algorithms have been adopted to train prediction models on the original metrics or distilled metrics. For learning on the original metrics, [4] used a Bayesian network to measure the importance of metrics regarding defectiveness, [16] proposed an ensemble system of seven common classifiers. Other studies try to refine metrics using deep learning [6], feature selection/weighting [17], and principal component analysis (PCA). Yang et al. applied a deep belief network (DBN) to select expressive features [18]. In [6], the metrics were mapped into latent representations by an autoencoder. The results demonstrate the effectiveness of the latent representations in comparison with the original metrics.

Nowadays, the application of deep learning to automatic feature extraction has achieved state-of-the-art performance in software defect prediction. Many deep neural networks have been developed to generate sophisticated features from programs’ representations. In Mou et al. [19] designed a tree-based convolutional network on abstract syntax trees (AST) obtained by parsing source files. Other studies generated sequences of AST tokens and trained prediction models using DBN [12], Long Short-Term Memory (LSTM) [13], Convolutional Neural Networks (CNNs) [20], bidirectional LSTM with the attention mechanism [21]. Since close to machine code, learning on assembly instructions resulted by program compilation have achieved high performance in defect prediction. In [14], convolutional neural networks were applied directly on assembly instruction sequences. To enrich input information, Phan et al. constructed control flow graphs (CFG) and then developed a graph-based neural network to generate defective features [22]. The experimental results on four datasets showed that assembly instruction-based methods significantly outperformed AST-based methods and other baselines.

The unbalanced data problem is frequently encountered in defect prediction tasks because the number of defective samples generally dominates that of clean samples. To prevent classifiers from biasing towards majority classes, different learning strategies have been proposed such as subsampling, cost-sensitive, and ensemble. Subsampling strategies aim to rebalance samples among classes prior to the model construction. The main approaches include 1) undersampling that eliminates majority class instances, 2) oversampling [23] that randomly replicates minority class instances, 3) SMOTE [24] that generates new minority class instances from nearest neighbors, and 4) hybrid methods that combine other techniques. Cost-sensitive learning treats the misclassification of classes with different costs [25]. Ensemble learning combines the predictions from multiple algorithms to make the decision [10].

The lack of annotated samples is one of the main challenges for defect prediction. To alleviate this issue, many strategies have been employed including transfer learning, self-supervised learning, masked language modeling. In [26], Phan et al. collected data from different sources to train a convolutional model and used it as a program feature extractor. In [27], Feng created a pretrained model, namely CodeBERT for programming and natural languages.

Previous studies, whether handcrafted or automatic, treated program components equally when learning features. Unlike, our methods direct the learner to smaller components to extract distinguishing features. During the feature generation, similar samples are assigned to a cluster regardless of being defective or clean. Then, we apply penalty functions to enlarge sample distances inside a cluster to distinguish, and shrink cluster centroids to reduce the variance.

III. THE PROPOSED MODEL

Figure 2 shows the proposed model architecture. The model is composed by 4 components including 1) a convolutional neural network, 2) an autoencoder, 3) a k-means cluster, 4) and a classifier. The convolutional network undertakes feature extraction from the input. The autoencoder and the k-means cluster assist the convolution in generating features that retain the important information, and are more distinguishable. The main task of defective probability prediction is performed by the classifier.

A. THE CONVOLUTIONAL NEURAL NETWORK

The model aims to predict software defects at file level. An original input is an assembly instruction sequence obtained by compiling a source file or decompiling an executable file. The sequence is then converted into numerical representations before feeding into the model. In the model, the convolutional neural network applies a set of convolutional and pooling layers to learn features from inputs.

Embedding layer provides numerical input representations for networks. Given an instruction sequence \( \{a_i\}_{i=1}^L \), the embedding matrix \( X \) is formed by mapping each item into a \( d \)-dimensional vector.

\[
X = e_{a_1} \oplus e_{a_2} \oplus \ldots \oplus e_{a_L}
\]

where \( X \in \mathbb{R}^{L \times d} \), \( e_{a_i} \in \mathbb{R}^d \) is the vector of instruction \( a_i \), and \( \oplus \) is the concatenation operator.

Convolutional layers use a set of filters to extract local features. In convolutional layers, each neuron is just connected with a sub-structure of the input. Filters consist of learnable parameters to define the connection weights and regions. Each filter aims to extract specific types of features. During the forward pass, the filter slides along the input
sequence and produces a feature map as follows:

\[ c_i = f(W_c \cdot e_{a_i+b} + b) \]  

(2)

where \( i = 1, \ldots, L, f \) is a non-linear activation function, \( h \) is the filter size, \( e_{a_i+b} = e_{a_i} \oplus e_{a_{i+1}} \oplus \ldots \oplus e_{a_{i+h-1}}, W \in \mathbb{R}^{h \times d} \) is the weights, and \( b \) is the bias. Intuitively, a feature map contains neurons that are arranged along the input sequence, connect to a local region, and share the same weights and bias. By the sharing weight mechanism, the filter can emphasize the features appearing in sub-structures at any position in the input data. Since instruction sequences are very long and defects just occur in small parts of programs, convolutional neural networks are employed instead of other architectures like long short-term memory (LSTM) [28], and transformers [29]. A filter with a size of \( h \) explores features for \( h \)-element regions. Stacking multiple convolutional layers allows the network to learn high-level abstract features and expand feature extraction regions.

A pooling layer is commonly stacked on each convolutional layer to perform dimension reduction. For high-dimensional data, downsampling helps to reduce model parameters, and hence to speed up computational time and control overfitting. In experimental datasets, a program may have up to 3,000 instructions resulting in a 3,000 embedding matrix. In addition, the convolution just transforms the input sequence, connect to a local region, and share the same weights and bias. By the sharing weight mechanism, the filter can emphasize the features appearing in sub-structures at any position in the input data. Since instruction sequences are very long and defects just occur in small parts of programs, convolutional neural networks are employed instead of other architectures like long short-term memory (LSTM) [28], and transformers [29]. A filter with a size of \( h \) explores features for \( h \)-element regions. Stacking multiple convolutional layers allows the network to learn high-level abstract features and expand feature extraction regions.

In pooling layers, each feature map is resized independently. Normally, a pooling layer splits a feature map into non-overlapping regions and then applies the pooling operation for such sub-regions. Two commonly used operations are max and average. For example, a max pooling with the filter size of 2 and the stride of 2, the feature map column \( C = \{c_i\} \) will be separated into two-value regions, and the greater one of each pair is selected. As a result, the pooled feature map has a size of \( L/2 \). In the training process, the backward pass simply routes the gradient to the highest value in the forward pass.

**B. THE AUTOENCODER**

An autoencoder consists of two segments, the encoder and the decoder (parts 1 and 3) that have symmetric structures.

The encoder maps input sequences \( X \) by function \( \phi \) to latent representations \( F \). Inversely, the decoder tries to recreate the original sequences \( X \) by function \( \psi \). The process can be formulated as follows.

\[ \phi : X \rightarrow F \]
\[ \psi : F \rightarrow X \]
\[ \phi, \psi = \arg\min_{\phi, \psi} \|X - (\phi \circ \psi)X\|^2 \]  

(3)

In the model, both the encoder and the decoder are built up from convolutional layers. A slight difference between the two architectures is that the pooling is used for dimension reduction in the encoder, while the decoder applies upsampling to expand feature maps.

The objective of training the autoencoder is to minimize reconstruction errors. The loss function for an instance \( x_i \) is as follows.

\[ L_{\text{reconst}} = L(x_i, x'_i) = \|x_i - x'_i\|^2 \]
\[ = \|x_i - \phi(\psi(x_i))\|^2 \]  

(4)

where \( x'_i \) is the reconstruction that has the same shape as \( x_i \).

**C. THE K-MEANS**

K-means is an unsupervised algorithm to partition data samples into disjointed clusters. Samples within a cluster are similar and dissimilar to those in other clusters. The similarity is evaluated based on a specific distance such as Euclidean, Manhattan and Minkowski.

Algorithm 1 indicates the steps of k-means. The inputs include data samples and the cluster number. The algorithm estimates cluster presentation points called centroids and assigns samples to clusters. Firstly, the centroids are randomly initialized. Next, the centroids and sample labels are recalculated repeatedly until meeting stopping criteria. Some criteria can be applied including 1) centroids do not change, 2) samples remain in the same clusters, 3) an iteration number.

In the model, the k-means cluster programs according to their contents. As mentioned in Section I (Figure 1), different implementations can be used to perform a task, and defects can occur in any program regardless of the implementations. To learn distinguishing features for defects,
we use k-means in conjunction with two constraints including 1) shrinking the centroids, and 2) stretching samples inside each cluster (Figure 3). The motivation behind shrinking is to reduce the variance, and stretching is to emphasize features to distinguish similar programs. The two constraints are implemented by the loss function as follows:

\[ L_{inter} = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d(c_i, c_j) \] (5)

where \( d(c_i, c_j) \) is the distance between two centroids \( c_i \) and \( c_j \).

\[ L_{intra} = -\frac{1}{k} \sum_{c=1}^{k} \sum_{x_i \in c} \sum_{x_j \in c, j \neq i} d(x_i, x_j) \] (6)

where \( c^{(i)} \) is the cluster of the sample \( x_j \).

D. THE CLASSIFIER

For predicting faulty code, the classifier is built based on the features extracted by the convolutional network. This part includes a global pooling and some fully connected layers. **Global pooling**: The last pooling layer in the part (1) is also high dimensional features. To select the most sophisticated features, a global pooling is applied to pool each feature map into a single value.

**Fully connected layers** contains neurons that fully connect to all neurons in the previous layer. The activation of the last layer is softmax to convert score values into distribution probabilities.

\[ \sigma(z_c) = \frac{e^{z_c}}{\sum_{i=1}^{K} e^{z_i}} \] (7)

where \( K \) is the number of target labels, \( c = 1, \ldots, K \), and \( z = (z_1, \ldots, z_K) \).

We use categorical cross-entropy to evaluate the classification loss. For an instance \( i \), the loss is computed as follows:

\[ L_{class} = L(y_i, \hat{y}_i) = -\sum_{c=1}^{K} y_{i,c} \log(p_{i,c}) \] (8)

where \( y_{i,c} \) is the indicator function that has value of 1 if and only if \( c \) is the correct target for the instance \( i \). \( p_{i,c} \) is the predicted probability for the class \( c \) as in Equation 7.

E. TRAINING THE MODEL

The **loss function** is a linear combination of the loss components in Equations 4, 5, 6, and 8.

\[ L = L_{class} + \lambda_{reconst} \ast L_{reconst} + \lambda_{inter} \ast L_{inter} + \lambda_{intra} \ast L_{intra} \] (9)

where \( \lambda \) factors adjust the contribution of each component.

**The training procedure**. All the components are jointly trained using the mini-batch gradient descent. Algorithm 2 presents the training procedure to find the best model parameters by minimizing the loss function on a dataset. Firstly, the parameters are randomly initialized and data samples are divided into batches with the same size. Next, training model is performed repeatedly on the whole dataset. At each epoch, the parameters are updated at every batch. Given a batch of samples \( X_b \), we pass into the model to compute the outputs of all components including input reconstructions \( \hat{X}_b \), latent representations \( L - b \), features \( F_b \), k-means centroids \( M_b \), sample clusters \( C_b \), and classification predictions \( \hat{Y}_b \). Finally, we compute the total loss and estimate the gradient to update the parameters.

IV. EXPERIMENTS

A. THE DATASETS

The proposed model is compared to state-of-the-art methods for software defect prediction on four datasets, namely
Algorithm 2: Mini-Batch Gradient Descent Algorithm

Input: Data samples \( x^{(i)}, i = 1..N \);
Learning rate \( \eta \), batch size \( b_s \);

Output: Model parameters \( \theta = \{ W, B \} \)
1 Randomly initialize \( \theta, \Delta \theta \leftarrow 0 \);
2 Partition \( x^{(i)} \) into \( b_s \)-sample batches \( X_{\text{batches}} = \{ X_b \} \);
3 for \( i \leftarrow 1 \) to epochs do
   for \( X_b \) in \( X_{\text{batches}} \) do
      \( \hat{X}_b, L_b = \text{Autoencoder}(X_b); \)
      \( F_b = \text{Global Pooling}(L_b); \)
      \( M_b, C_b = \text{k_means_cluster}(F_b); \)
      \( \hat{Y}_b = \text{classifier}(F_b); \)
      \( \text{Loss} = L_{\text{class}}(Y_b, \hat{Y}_b) + \lambda_{\text{inter}} * L_{\text{inter}}(M_b) + \lambda_{\text{intra}} * L_{\text{intra}}(C_b) + L_{\text{reconst}}(X_b, \hat{X}_b); \)
      \( \Delta \theta \leftarrow \text{evaluate gradient}(\text{Loss}, \theta); \)
      \( \theta \leftarrow \theta - \eta \Delta \theta; \)
   end
3 end

V. A. Phan: Learning Stretch-Shrink Latent Representations With Autoencoder and K-Means for Software Defect Prediction

SUMTRIAN, FLOW016, MNMX and SUBINC published in [14]. The comparisons are performed to different criteria including performance measures and the quality of generated features. Each dataset consists of C/C++ implementations for solving a problem on CodeChef, a site for learning programming languages. The four problems are described as follows:

- **SUMTRIAN** (Sums in Triangle): finding the largest of the sums of numbers on the path from the top towards the base in a \( n \)-row lower triangular matrix. On each path, the next number is located directly below or below and one place to the right.
- **FLOW016**: finding the greatest common divisor (GCD) and the least common multiple (LCM) of two integers.
- **MNMX**: finding the minimum sum of costs to convert the array into a single element by the operation of removing the larger of two adjacent integers. The cost of each operation is the smaller number.
- **Given an array \( A \) of \( N \) elements, count the number of non-decreasing subarrays of array \( A \)**

A non-defective program must produce the correct output and satisfy the problem’s constraints such as running time, and memory usage. Clean programs are labelled as 0, while others are tagged according to defective types including 1- time limit excess, 2- incorrect output, 3- interrupted execution, and 4- failed compilation.

All programs are transformed into assembly code by g++ compiler. The assembly code is selected as the model input because it is close to the machine code. Table 1 presents data statistics on the four experimental datasets. Remarkably, data samples among classes are greatly unbalanced wherein the ratios of several classes are just 1-3%. To train models, each dataset is split by the ratio of 3:1:1 for training, validation, and test. In which, these subsets are used to adjust the trainable parameters, select the best point, and evaluate the model performance.

### Table 1. Data statistics on the four datasets. maxlength is the length of the biggest program.

| Dataset       | Max length | #samples | ratio of samples (%) |
|---------------|------------|----------|----------------------|
| FLOW016       | 1,246      | 10,648   | 32.61                |
|               |            |          | 39.12                |
|               |            |          | 2.17                 |
|               |            |          | 22.24                |
|               |            |          | 3.87                 |
| MNMX          | 3,073      | 8,745    | 58.97                |
|               |            |          | 35.14                |
|               |            |          | 2.16                 |
|               |            |          | 1.29                 |
|               |            |          | 2.44                 |
| SUBINC        | 1,245      | 6,484    | 50.32                |
|               |            |          | 41.41                |
|               |            |          | 3.18                 |
|               |            |          | 1.51                 |
|               |            |          | 3.58                 |
| SUMTRIAN      | 2,905      | 21,187   | 43.10                |
|               |            |          | 32.79                |
|               |            |          | 1.98                 |
|               |            |          | 12.75                |
|               |            |          | 9.38                 |

### Table 2. The confusion matrix.

| Predicted | +1 (Positive) | -1 (Negative) |
|-----------|---------------|---------------|
| Actual     | TP            | FN            |
| +1         | TP            | FN            |
| -1         | PP            | TN            |

### B. BASELINES AND SETTINGS

To evaluate the performance, the proposed model is compared with state-of-the-art approaches for sequence learning including convolutional neural networks (CNN) [30], long short-term memory (LSTM) with/without attention mechanism [31], transformers [32], and graph-based networks [22]. The hyper-parameter settings for the networks are as follows:

- **LSTM** and **LSTM with attention (AttLSTM)**. The number of hidden state is 128.
- Transformer. The head is 2, and the hidden state is 128.
- **CNN** includes two convolutional layers wherein each is followed by a pooling. The filters numbers are 50, and 100. The filter size is set to 2.
- **DGCNN** has the configuration as in [22], consisting of two graph convolutional layers with 100 and 600 filters, a fully connected layer with 600 neurons, and a softmax.

### C. EVALUATION MEASURES

The approaches are evaluated according to popular performance measures including the classification accuracy, F1 and AUC (the area under the receiver operating characteristic - ROC). The accuracy, and F1 are computed from the confusion matrix. Considering binary classification problems wherein samples are tagged either +1 (positive) or -1 (negative), the confusion matrix is constructed as in Table 2. \( TP \) and \( FN \) are the total positive samples being predicted as positive and negative. Similarly, \( TN \) and \( FP \) are the total negative samples being predicted as negative and positive.

**Classification accuracy:**

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}
\]

**The F1 measure:**

\[
F1 = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{11}
\]

where \( \text{Precision} = \frac{TP}{TP + FP} \) and \( \text{Recall} = \frac{TP}{TP + FN} \)
TABLE 3. The performance comparison among approaches in terms of Accuracy and F1.

| Approach   | FLOW016 | MNMX | SUBINC | SUMTRIAN |
|------------|---------|------|--------|----------|
|            | Acc.    | F1   | Acc.   | F1       | Acc.    | F1   |
| LSTM       | 57.2 %  | 0.29 | 50.2 % | 0.19     | 63.2 %  | 0.36 | 50.0 % | 0.32 | 52.5 % | 0.42 | 48.7 % | 0.73 | 53.2 % | 0.51 | 46.2 % | 0.33 |
| AttLSTM    | 71.5 %  | 0.31 | 70.8 % | 0.15     | 77.2 %  | 0.62 | 74.0 % | 0.66 | 58.9 % | 0.62 | 56.5 % | 0.48 | 65.0 % | 0.62 | 63.5 % | 0.11 |
| Transformer | 60.9 %  | 0.31 | 59.4 % | 0.44     | 68.7 %  | 0.34 | 66.2 % | 0.53 | 58.5 % | 0.46 | 54.1 % | 0.73 | 57.5 % | 0.72 | 51.8 % | 0.74 |
| CNN        | 72.5 %  | 0.23 | 71.3 % | 0.15     | 81.9 %  | 0.06 | 80.0 % | 0.21 | 70.9 % | 0.71 | 69.9 % | 0.32 | 67.9 % | 0.34 | 66.3 % | 0.42 |
| DGCNN      | 72.4 %  | 0.59 | 71.6 % | 0.51     | 82.7 %  | 0.56 | 80.9 % | 0.57 | 72.4 % | 1.36 | 71.3 % | 1.03 | 69.6 % | 0.42 | 68.1 % | 0.84 |
| AE_Kmeans  | 73.9 %  | 0.67 | 72.9 % | 0.56     | 82.9 %  | 1.55 | 80.9 % | 1.98 | 74.0 % | 0.67 | 73.0 % | 0.81 | 71.3 % | 0.46 | 70.8 % | 0.46 |

AUC that shows the degree to distinguish among classes is a good measure in the case of unbalanced data. For unbalanced data, learning algorithms normally bias towards the majority classes and ignore the minorities. In many practical problems, detecting the minorities, e.g., defects, is more significant than that of the majorities. The higher the AUC is, the better distinguishing ability the model has. AUC is the area under the ROC curve that plots the true positive rate \( TPR = \frac{TP}{P} \) and the false positive rate \( FPR = \frac{FP}{N} \) at different thresholds.

For multi-class problems, the metrics can be estimated by some methods such as 1) micro - calculating metrics globally by counting prediction outcomes, 2) macro - calculating metrics for each class, and find their unweighted mean, 3) weighted - calculating metrics for each class, and take the average weighted by the number of samples.

V. RESULTS AND DISCUSSION

Table 3 compares the approaches in terms of accuracy and F1. AE_Kmeans outperforms other baselines on four experimental datasets. It is worth noting that although the same architecture, AE_Kmeans overcomes CNN thanks to the k-means support. In comparison with DGCNN, a recent state-of-the-art approach on these datasets, AE_Kmeans improves accuracy remarkably on FLOW016, SUBINC, and SUMTRIAN. Because of very long programs (as in Table 1), sequentially compressing to generate features is inefficient. As a proof, LSTM obtains a low performance, although it is a powerful model for various sequence processing tasks. The application of the attention mechanism in AttnLSTM and Transformer to reduce long-term dependencies helps to boost LSTM significantly. Since defects just exist in small parts of the programs, extracting local features is beneficial. For this reason, both sequence-based and graph-based convolutional neural networks achieve higher performance than those of sequential compression approaches. In the proposed network, the autoencoder guarantees that generated features contain important information of programs; the k-means groups similar programs and focuses on different small parts to distinguish. Thus, the use of these components facilitates the feature generation by the CNN. Above analyses have indicated the importance of the k-means and the autoencoder in the model.

Table 4 compares AE_Kmeans and DGCNN on minority classes by AUC. For unbalanced data, learning models normally have inaccurate predictions on minority classes, while detecting them is more preferable. Taking the defect prediction into account, although defective programs are minor, recognizing them is more significant than that of clean. Thus, AUC that indicates the ability to distinguish among classes is an important measure to evaluate models. From Tables 3 and 4, the proposed model optimizes accuracy without a decrease of the distinguishing ability despite highly unbalanced datasets. With rich information of the input representations, DGCNN outperforms other network architectures including CNN, LSTM and Transformer. However, AE_Kmeans achieves higher not only accuracy and F1 but also AUC than that of DGCNN in most cases. As mentioned in Section I, defects may exist in any program regardless of coding styles. Distinguishing similar codes with different defective types is difficult. The high performance of AE_Kmeans has indicated that dividing programs according to code similarities and stretching within classes are beneficial to defect prediction.

Figure 4 depicts the ROC curves generated by the two best models on SUBINC. For majority classes such as 0 and 1,
TABLE 4. The performance comparison on minority classes according to AUC.

| Dataset     | class 2             | class 3             | class 4             |
|-------------|---------------------|---------------------|---------------------|
|             | AE_Kmeans DGCNN     | AE_Kmeans DGCNN     | AE_Kmeans DGCNN     |
| FLOW016     | 61.48±1.82 59.34±2.65 | 95.86±0.28 95.33±0.28 | 77.52±1.37 72.67±3.31 |
| SUBNC       | 60.34±1.25 58.31±3.81 | 70.97±3.27 77.35±3.67 | 75.74±1.9 78.63±2.46 |
| MNMX        | 61.71±1.94 65.6±2.01 | 77.87±2.17 74.56±5.14 | 85.46±2.6 85.4±1.86 |
| SUMTRIAN    | 62.01±2.33 58.6±3.28 | 91.34±0.69 91.67±0.71 | 85.88±0.18 84.53±1.29 |

FIGURE 5. A visualization of features generated by AE_Kmeans and DGCNN on FLOW016 (5a, 5b) and MNMX (5c, 5d).

both models achieve high performance with the AUC higher than 80%. When increasing the decision threshold, the curves run along the vertical and upper horizontal axes. However, DGCNN obtains a low performance on minority classes, especially class 2 with the sample ratio of 3.18%. The ROC curve approaches the diagonal line and AUC is around 53%. This performance is just equivalent to a random classifier. In contrast, the ROC curve produced by AE_Kmeans is much higher than the diagonal line and the AUC reaches 61%.

To verify the effects of k_means and autoencoder on the feature learning, programs’ features generated by the approaches are visualized for analysis. Figure 5 shows visualizations of FLOW016 and MNMX for AE_Kmeans and DGCNN. The features are linearly transformed into a two dimensional space using the principal component analysis method (PCA) for the visualization. For DGCNN, programs are scattered on the feature space (subfigures 5b and 5d). With the support of k-means, the features generated by the proposed model are separated into different small clusters regarding coding styles. The stretch penalty function to enlarge samples in each cluster allows the model to detect defects in similar programs.

VI. CONCLUSION

This paper proposed a new deep learning model consisting of a CNN, an autoencoder, and a k-means for software defect prediction. One of the main reasons leading to the challenges in these problems is that similar programs may belong to different defective classes. Common learning algorithms may tag similar programs into one defective class. To address the issue, the proposed model uses the autoencoder to support the CNN to retain important information, and the k-means with two penalty functions to group similar programs and try to distinguish them. The experiments on four datasets have shown the efficiency of our method for software defect prediction.

REFERENCES

[1] H. Krasner, “The cost of poor quality software in the US: A 2018 report,” Consortium IT Softw. Qual., Tech. Rep., Sep. 2018. [Online]. Available: https://www.it-cisq.org/
[2] B. Sheehan, F. Murphy, M. Mullins, and C. Ryan, “Connected and autonomous vehicles: A cyber-risk classification framework,” Transp. Res. A, Policy Pract., vol. 124, pp. 523–536, Jun. 2019.
[3] M. Cummings and D. Britton, “Regulating safety-critical autonomous systems: Past, present, and future perspectives,” in Living With Robots. Amsterdam, The Netherlands: Elsevier, 2020, pp. 119–140.
[4] A. Okutan and O. T. Yıldız, “Software defect prediction using Bayesian networks,” Empirical Softw. Eng., vol. 19, no. 1, pp. 154–181, 2014.
V. A. Phan: Learning Stretch-Shrink Latent Representations With Autoencoder and K-Means for Software Defect Prediction

[5] S. S. Rathore and S. Kumar, “A decision tree logic based recommendation system to select software fault prediction techniques,” Computing, vol. 99, no. 3, pp. 255–285, Mar. 2017.

[6] H. Tong, B. Liu, and S. Wang, “Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning,” Inf. Softw. Technol., vol. 96, pp. 94–111, Apr. 2018.

[7] C. Manjula and L. Florence, “Deep neural network based hybrid approach for software defect prediction using software metrics,” Cluster Comput., vol. 22, no. 4, pp. 9847–9863, 2019.

[8] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, “Software defect prediction based on kernel pca and weighted extreme learning machine,” Inf. Softw. Technol., vol. 106, pp. 182–200, Dec. 2019.

[9] Q. Song, Y. Guo, and M. Shepperd, “A comprehensive investigation of the role of imbalanced learning for software defect prediction,” IEEE Trans. Softw. Eng., vol. 45, no. 12, pp. 1253–1269, Dec. 2019.

[10] S. Huda, K. Liu, M. Abdelrazek, A. Ibrahim, S. Alyahya, H. Al-Dossari, and S. Ahmad, “An ensemble oversampling model for class imbalance problem in software defect prediction,” IEEE Access, vol. 6, pp. 24184–24195, 2018, doi: 10.1109/access.2018.2817572.

[11] O. F. Arar and K. Ayan, “Software defect prediction using cost-sensitive neural network,” Appl. Soft Comput., vol. 33, pp. 263–277, Aug. 2015.

[12] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep semantic feature learning for software defect prediction,” IEEE Trans. Softw. Eng., vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[13] H. Liang, Y. Yu, L. Jiang, and Z. Xie, “SemiL: A semantic LSTM model for software defect prediction,” IEEE Access, vol. 7, pp. 83812–83824, 2019.

[14] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, “Convolutional neural networks over control flow graphs for software defect prediction,” in Proc. IEEE 29th Int. Conf. Tools Artif. Intell. (ICTAI), Nov. 2017, pp. 45–52.

[15] Z. Han, X. Xie, A. E. Hassan, D. Lo, J. Yin, and X. Yang, “Perceptions, expectations, and challenges in defect prediction,” IEEE Trans. Softw. Eng., vol. 46, no. 11, pp. 1241–1266, Nov. 2020.

[16] I. H. Laradji, M. Alshayba, and L. Ghouti, “Software defect prediction using ensemble learning on selected features,” Inf. Softw. Technol., vol. 58, pp. 388–402, Feb. 2015.

[17] H. Wang, T. M. Khoshgoftaar, and A. Napolitano, “A comparative study of ensemble feature selection techniques for software defect prediction,” in Proc. 9th Int. Conf. Mach. Learn. Appl., Dec. 2010, pp. 135–140.

[18] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” in Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur., Aug. 2015, pp. 17–26.

[19] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in Proc. 30th AAAI Conf. Artif. Intell., 2016, pp. 190–210.

[20] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS), Jul. 2017, pp. 318–328.

[21] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Software defect prediction via attention-based recurrent neural network,” Sci. Program., vol. 28, pp. 1–14, Apr. 2019.

[22] A. V. Phan, M. L. Nguyen, Y. L. H. Nguyen, and L. T. Bui, “DGNN: A convolutional neural network over large-scale labeled graphs,” Neural Netw., vol. 108, pp. 533–543, Dec. 2018.

[23] R. Yedida and T. Menzies, “On the value of oversampling for deep learning in software defect prediction,” IEEE Trans. Softw. Eng., vol. 48, no. 8, pp. 3103–3116, Aug. 2022.

[24] L. Pelayo and S. Dick, “Applying novel resampling strategies to software defect prediction,” in Proc. Ann. Meeting North Amer. Fuzzy Inf. Process. Soc. (NAFIPS), Jun. 2007, pp. 69–72.

[25] M. T. Kai, “An instance-weighting method to induce cost-sensitive trees,” IEEE Trans. Knowl. Data Eng., vol. 14, no. 3, pp. 659–665, Dec. 2002.

[26] A. P. Viet, K. D. T. Nguyen, and L. V. Pham, “Transfer learning for predicting software faults,” in Proc. 11th Int. Conf. Knowl. Syst. Eng. (KSE), Oct. 2019, pp. 1–6.

[27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” 2020, arXiv:2002.08155.

[28] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput., vol. 9, no. 8, pp. 1735–1780, 1997.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in Proc. Adv. Neural Inf. Process. Syst., vol. 30, 2017, pp. 1–5.

[30] A. V. Phan and M. Le Nguyen, “Convolutional neural networks on assembly code for predicting software defects,” in Proc. 21st Asia–Pacific Symp. Intell. Syst. (IES), Nov. 2017, pp. 37–42.

[31] G. Liu and J. Guo, “Bidirectional LSTM with attention mechanism and convolutional layer for text classification,” Neurocomputing, vol. 337, pp. 325–338, Jun. 2019, doi: 10.1016/j.neucom.2019.01.078.

[32] P. Li, P. Zhong, K. Mao, D. Wang, X. Yang, Y. Liu, J.-X. Yin, and S. See, “Act: An attentive convolutional transformer for efficient text classification,” in Proc. AAAI Conf. Artif. Intell., vol. 35, 2021, pp. 13261–13269.

VIET ANH PHAN received the B.Sc. degree in information technology and the M.Sc. degree in computer science from Le Quy Don Technical University, Vietnam, in 2008 and 2013, respectively, and the Ph.D. degree (Hons.) in computer science from the Japan Advanced Institute of Science and Technology (JAIST), in 2018.

He has participated as a key member in various research and industrial projects related to database management systems, and applications of machine learning to image and language processing. Currently, he is a Lecturer with Le Quy Don Technical University (LQDTU). He has published refereed academic papers/articles on high-quality conferences, such as ICTAI, IDEAL, and IES and journals, such as Applied Intelligence, Data & Knowledge Engineering, and Neural Networks. He is a Reviewer of the KSE Conference and journals of Data & Knowledge Engineering, and VNU Journal of Science: Computer Science and Communication Engineering. His research interests include machine learning, software engineering, evolutionary computation, and deep learning. He received the Outstanding Student Award from JAIST.