2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-1594-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSRE59848.2023.00070

A Two-Stage Framework for Ambiguous Classification in Software Engineering

Jiaying Li†, Yan Lei‡*, Shanshan Li†*, Haifang Zhou†, Yue Yu†, Zhouyang Jia†, Yingwei Ma†, Teng Wang†
†National University of Defense Technology, China
Email: {lijiaying, shanshanli, haifang_zhou, yuyue, jiazhouyang, myw, wangteng13}@nudt.edu.cn
‡Chongqing University, China
Email: yanlei@cqu.edu.cn

***Abstract*—Classification tasks are prevalent and play a crucial role in the field of software engineering. However, when two classes exhibit similar features at the class level, the classification model is prone to misclassification, which we refer to as ambiguous classification, and the corresponding classes as ambiguous classes. Ambiguous classification may impact the security and reliability of software engineering classification systems.**

To correct ambiguous classification, we propose a two-stage framework. Our key insight is to combine two different classification models and utilize their complementary knowledge to maximize the classification ability of the two-stage framework. Specifically, we identify ambiguous classes according to the confusion matrix of the original model. Then, we construct a two-stage model, where the first stage utilizes the original model and the second stage utilizes a different model trained on the same dataset. The second-stage model is responsible for reclassifying the samples that are predicted as ambiguous classes by the first-stage model. We evaluate our method on two software engineering tasks. Experimental results indicate that our method can effectively correct ambiguous classification and achieve a relative improvement of 19.8% in F1-score for ambiguous classes.

***Index Terms*—software reliability, classification task, software repair**

I. Introduction

Classification is a prominent research area in computer science encompassing various fields such as computer vision, speech recognition, natural language processing, and other fields [1]–[8]. In the field of software engineering, classification tasks are equally prevalent, examples of which include vulnerability detection, software requirement classification, code author attribution, and algorithm classification [9]–[16]. With the emergence and development of Deep Neural Networks(DNNs), automated classification methods are gradually applied in software engineering-related classification tasks, such as classification models based on LSTM [17] and BERT [18], which further improve the accuracy and efficiency of classification models [19]–[22].

However, despite the success of automated classification methods in software engineering-related classification tasks, some unexpected classification errors are still possible. For instance, when the BERT-based classification model is used to classify vulnerability code, the probability of misclassifying the class **Input validation error** as the class **Privilege escalation** reaches 35%, which is significantly higher than the misclassification rate of the models for other classes, as shown in Fig. 1.

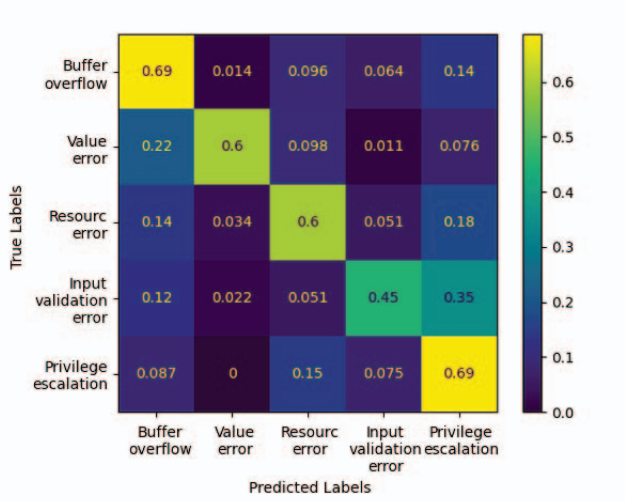


Fig. 1. A classification model is prone to misclassify class Love as class Joy.

We studied classification errors in software engineering and found that these classification errors typically manifest as models confusing two classes, resulting in a high misclassification probability. Different from the misclassification of a single instance concerned by existing studies [23]–[27], our research focuses on the misclassification of two classes by the models at the class level.

We refer to these classification errors in software engineering classification models as *ambiguous classification* and the corresponding classes as *ambiguous classes*. Essentially, ambiguous classification occurs when the model struggles to distinguish between two classes due to their similar class-level features, which we call *ambiguous features*. These features make it challenging for the model to differentiate between the classes effectively. It's important to note that ambiguous features affect the entire class, unlike the unique features of individual samples. This means that when two classes share ambiguous features, all samples within those classes are prone to be misclassified by the model. However, not every sample will be misclassified. For instance, a classification model may easily misclassify class *Input validation error* and class

*Corresponding authors.

2332-6549/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSRE59848.2023.00070

Privilege escalation, while some individual code that falls under *Input validation error* can still be correctly predicted.

Similar to software errors that can lead to serious consequences, classification errors present a risk to the security and reliability of classification systems. Existing research focuses on addressing misclassification at the instance level and dataset level. It includes solving the instance-level error of classification models based on adversary training, so as to improve the robustness of the models [23]–[29]. And through data augmentation and retraining technology to improve the overall accuracy of the model on the dataset [30]–[34]. However, there is a gap in research on correcting the ambiguous classification error in models, which is caused by the confusion of models at the class level.

In this paper, we propose a two-stage framework to correct ambiguous classification errors in the software engineering classification model, which can be seen as an Ensemble Learning method [35]. Our key insight is that different models may focus on different aspects of the data. By combining two distinct models, we can leverage their complementary knowledge to maximize the classification ability of our two-stage framework.

Specifically, we first identify the ambiguous classes according to the confusion matrix [36] of the original model on the test set. When the confusion matrix shows that the model has much higher misclassification probabilities in some classes than in others, we believe there is an ambiguous classification in the classification model, and the corresponding two classes are ambiguous classes. Next, we construct a two-stage classification framework. The first stage employs the original model, while the second stage uses a different model trained on the same dataset. We believe that the samples predicted by the first-stage model as an ambiguous class and at the decision boundary are more likely to be misclassified. Therefore, we set an uncertainty threshold to filter these samples and reclassify them by the second-stage model.

We conduct experiments to evaluate the effectiveness of our method on two software engineering tasks, i.e., software engineering emotion recognition and vulnerability classification. The experimental results show that our two-stage framework can effectively correct ambiguous classification errors without affecting the performance of other classes, and achieve a relative improvement of 19.8% in F1-score for ambiguous classes. Furthermore, we conduct an ablation study to examine the contributions of the first-stage model and the second-stage model, respectively. The results indicate that these models can focus on different features of the same data. When combined in the two-stage framework, their performance surpasses that of using them separately.

The main contributions of this paper can be summarized as follows:

- We define ambiguous classification in classification models applied to software engineering. To the best of our knowledge, we are the first to define this type of error in software engineering.
- We propose a two-stage framework designed to correct ambiguous classification errors. This framework leverages the strengths of multiple classification models and combines their knowledge to improve model performance.
- We evaluate our method on two software engineering-related classification tasks. The results indicate that our method can effectively correct ambiguous classification.

The rest of the paper is organized as follows. Section II introduces the background knowledge. Section III depicts our method. Section IV presents the experimental design. Section V discusses our experimental results on two tasks. Section VI is about discussion. Section VII summarizes related work. Finally, Section VIII concludes the whole study.

II. Background

A. Automated Classification Methods in Software Engineering

In software engineering, classification tasks cover crucial stages such as requirements analysis, software design, development, testing, and maintenance, which are closely related to the security and reliability of software [22]. In recent years, with the emergence and development of Deep Neural Networks (DNNs) and Data Mining technologies, automated classification methods are gradually applied in software engineering-related classification tasks. Among them, DNNs-based classification models are a common and widely used method, such as LSTM [17] and BERT [18].

DNNs are widely used machine learning models that simulate the interconnection structure between neurons in the human brain, consisting of a series of connected computing units called neurons. These neurons are arranged in different layers in order and transmit information through the transmission of signals between connections. Training the DNNs requires updating the weight parameters of the connections to enable the neurons in the last layer to produce the expected output. In the classification task, the goal of a DNN is to learn decision boundaries between different classes. Ideally, all instances belonging to the same class should be classified consistently, while instances from different classes should be correctly distinguished [37]. However, in high-dimensional feature spaces, if the distance between classes is small or below a predetermined threshold, DNNs may not be able to fully distinguish them.

B. Testing and Repairing of Classification Models

Existing research has made many efforts in testing and repairing classification models based on DNNs. In terms of testing, lots of testing methods based on neuron coverage have been proposed [38]–[43]. These testing tools use neuron coverage as a metric to detect instance level errors in models.

Regarding repairing, existing research focuses on improving the robustness of models to adversarial instances, as well as improving overall accuracy through data augmentation and retraining techniques [23]–[25], [28], [30]–[32], [34], [44]. For example, Ren et al. [33] proposed FSGMix, which maximizes the utilization of limited failed instances and enhances training

data through guidance from failed instances. Gu et al. [29] proposed GCN, which uses hierarchical contraction penalty and smoothness penalty terms to improve the robustness of models to adversarial samples by smoothing the model output mechanism.

While existing research primarily addresses instance-level and dataset-level errors in classification models, we focus on the reduction of ambiguous classification in software engineering, which is a class-level error.

C. Confusion Matrix and Uncertainty Strategy

Confusion matrix [36], a commonly used method in machine learning to evaluate the performance of classification models, which is suitable for binary and multi-class classification tasks. It is a two-dimensional matrix that intuitively displays the correspondence between the predicted results of the classification model on the test set and the actual labels. The values on the main diagonal of the matrix indicate correct classifications, whereas larger values indicate better performance of the model on the corresponding class. The values outside the main diagonal represent misclassifications, where larger values indicate higher probabilities of misclassification and poorer performance on the corresponding class.

Active learning is a subfield of machine learning, more generally, artificial intelligence [45]. The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer labeled training instances if it's allowed to select the most valuable training data from which it learns. The selection strategy of hard samples is an important step in active learning [45]. Best-vs-Second-Best (BvSB) is a widely used uncertainty sampling strategy, which compares the prediction probabilities of the model for the two most likely labels [46]. Samples with smaller probability differences are considered more uncertain. In this paper, we use a selection strategy similar to BvSB, but we do not use these samples for retraining. Instead, we identify potential misclassified samples for reclassifying. Through this approach, we can more specifically handle samples with a high risk of error, improving the performance and robustness of the model.

D. Ensemble Learning Methods

Ensemble Learning [35] is a machine learning method that involves training multiple classifiers and combining them to achieve improved overall predictive performance. By aggregating the predictions of multiple classifiers, ensemble learning can reduce the risk of overfitting and enhance the model's robustness and accuracy.

Classic ensemble learning methods include Bagging [47], Boosting [48], Stacking [49] and Delegation [50]. In Bagging, multiple classifiers are trained by randomly sampling the dataset with replacement, and then their results are combined through voting or averaging. Boosting, sequentially trains a series of weak classifiers, where each learner tries to correct the errors made by its predecessor, ultimately forming a strong learner. Stacking involves building a second-stage meta-learner that learns from the outputs of the base classifiers and produces the final prediction. Delegation removes the examples which are classified with high confidence and leaves the examples which are classified with lower confidence for subsequent iterations. Delegation is a layered problem-solving method similar to the concept of Defence-in-Depth (DiD) in the field of network security [51]. DiD requires the application of security at multiple levels, and its working principle is to provide different types of protection for each level to ensure that it becomes the best means to prevent attacks.

Compared to other ensemble learning methods, Delegation offers improved efficiency and interpretability, providing the possibility to simplify the overall multi-classifier by removing delegated parts. Our proposed two-stage method shares similarities with Delegation as both are sequential approaches triggered by confidence to classify examples in the second stage. However, they also differ in several aspects. Firstly, the two-stage method is more flexible than Delegation because the two classifiers used in the two-stage method are different and have different knowledge. In Delegation, all classes are the same. For example, if a decision tree is used as the base classifier, then the entire classifier is a decision tree. Secondly, in the Delegation method, the second stage classifier only uses samples with insufficient confidence in the first stage classifier for training, which can easily lead to overfitting. However, in the two-stage method proposed in this paper, the second-stage classifier is trained using a complete training set.

III. Methodology

In this section, we will introduce the overall of our two-stage framework, and then describe the details of each part.

A. Overall Architecture

First, we identify the ambiguous classes according to the confusion matrix of the original model on the test set. Then, we construct a two-stage framework. The first stage employs the original model and the second stage utilizes a different model trained on the same dataset. We filter out the samples that are predicted to be ambiguous class and located at the decision boundary according to the prediction probability of the first-stage model. Then, we use the second-stage model to reclassify these filtered samples, so as to correct the potential ambiguous classification error in the original model. Fig. 2 shows the overall structure of our proposed two-stage framework, consisting of three parts:

- Ambiguous class identification: Identify the ambiguous classes that lead to ambiguous classification error through the confusion matrix output from the original model.
- Boundary Sample Filtering: By setting the uncertainty threshold to filter out the samples located at the ambiguous class decision boundary, which are likely to be misclassified by the first-stage model.
- Second-stage Model: Reclassify the selected decision boundary samples of the ambiguous class, in order to correct the ambiguous classification error in the first-stage model through knowledge complementarity.

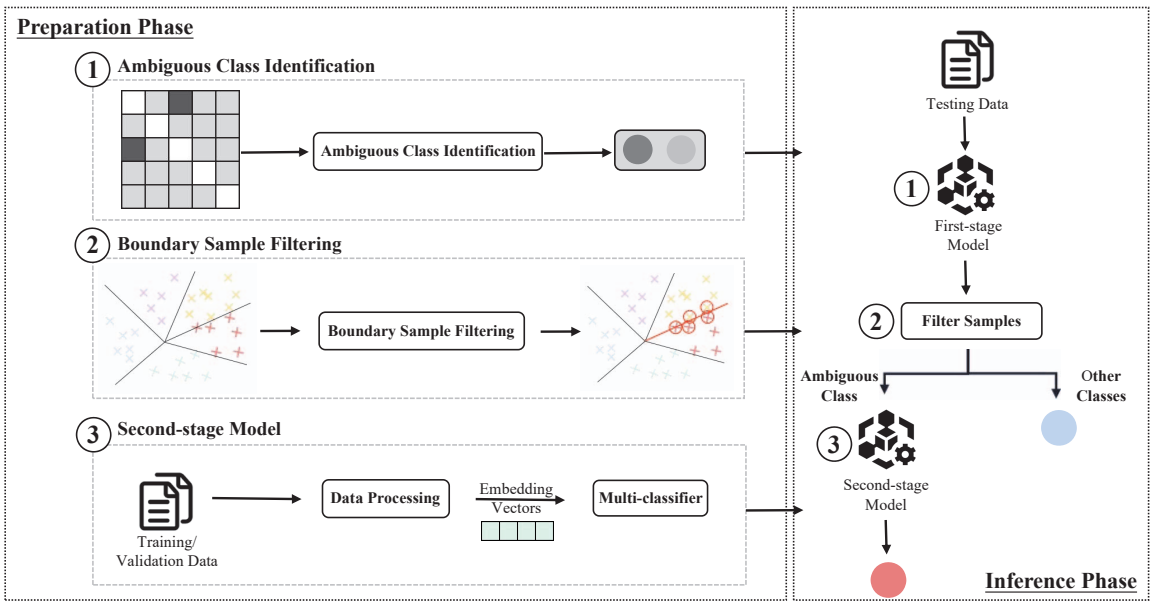


Fig. 2. Overall structure of the two-stage framework.

B. Ambiguous Class Identification

According to the definition, ambiguous classes appear in pair, which we refer to as ambiguous class pair. We use the confusion matrix output from the original model on the test set to determine the ambiguous classes and express them as pair. For multi-classification tasks, the confusion matrix is an $N \times N$ matrix, where N represents the number of classes, and shows the difference between the predicted and true label of a model. For a given confusion matrix, we calculate the third quartile of all off-diagonal values and select twice the number as the threshold θ_1. When a value in the confusion matrix is higher than θ_1, then the two corresponding classes on both sides of the diagonal are ambiguous classes, which can be represented by *(ambiguous class A, ambiguous class B)*. In statistics, quartiles can help us identify outliers in a data set because they can better resist the impact of extreme values. In this scenario, using twice the third quartile as the threshold θ_1 can help us find values that are significantly higher than most of the proportion of wrong predictions, and thus find the ambiguous classes. The calculation steps are shown in equations 1 and 2.

$$\theta_1 = 2 \cdot Quartile(CM) \tag{1}$$

$$C_i = \begin{cases} 1 & p_{ij} > \theta_1 \\ 0 & otherwise \end{cases} \tag{2}$$

In equations 1 and 2, CM is the confusion matrix of the original model on the test set. $Quartile$ is used to calculate the third quartile and get θ_1. The probability p_{ij} in the confusion matrix represents the probability that *Class i* is misclassified into *Class j*. And, we use 1 for ambiguous classes and 0 for other classes.

C. Boundary Sample Filtering

According to the uncertainty of prediction, we select the samples that are at the decision boundary because they may be misclassified, and then reclassify these samples in the second stage. Firstly, we consider the prediction class of the first-stage model. If the prediction class belongs to the ambiguous classes, we further calculate the difference δ between the prediction probabilities corresponding to the two classes in the ambiguous class pair. This difference reflects the model's confidence in the prediction. The smaller the difference, the lower the model's confidence in prediction, and vice versa. Next, we set a threshold called uncertainty threshold θ_2. When the difference in prediction probability δ is less than this threshold θ_2, we believe that the model has low confidence in the prediction and may experience classification error. Such samples are considered to be near the decision boundary of the ambiguous class and need to be reclassified in the second stage. On the contrary, if the difference in prediction probability is greater than this threshold θ_2, it indicates that the model is relatively confident in its prediction and does not need to enter the second stage for further classification.

D. Second-stage Model

The second-stage model is used to reclassify the selected samples. We will introduce the selection and construction of the second-stage model in the following steps.

1) Model Selection in the Second Stage: We establish a model recommendation library, including models that are currently widely used and perform well. Based on the type of input data and the selection of the first-stage model, we form a model recommendation list as shown in Table I.

We drew on the research conclusions of Minaee et al [52]. Minaee et al. reviewed over 150 DNNs-based text

TABLE I
MODEL RECOMMENDATIONS FOR SOFTWARE ENGINEERING-RELATED CLASSIFICATION TASKS.

Task Type	First-stage Model	Second-stage Model
Natural Language Related	ML, RNN, LSTM, BERT	BERT, RoBERTa
Source Code Related	ML, LSTM, GGNN, CodeBERT	CodeBERT GraphCodeBERT

classification models proposed in recent years and suggested that pre-trained models could be prioritized when selecting classification models. For a given task, we follow the following principles to select the second-stage model:

- If the first stage is a machine learning model or a general DNNs-based model, then the second stage selects a pre-trained model in the corresponding domain.
- If the first stage is a pre-trained model, then the second stage selects a model with a different structure and is pre-trained on different datasets.

2) Data Processing: Building upon the preprocessing steps of the baseline method, we need to tokenize the raw data and convert them into vectors to enhance the second-stage model's understanding and exploration of the input data.

To enable the model to comprehend the data, we need to encode the token sequence into vectors, thereby transforming unstructured data into structured data. Encoding vectors involve representing each token as a numerical vector to facilitate model processing and analysis. These word vectors are then aggregated into sentence vectors. Finally, input the sentence vector into the model.

In our framework, we adopt a method of adapting to the model for tokenization and vector transformation. For BERT [18], we use BertTokenizer for tokenization. The BertTokenizer consists of the BasicTokenizer and the WordpieceTokenizer. The BasicTokenizer preprocesses the input data by removing special characters and applying case normalization. The WordpieceTokenizer further splits the tokenized tokens into smaller subwords. For RoBERTa [53] and CodeBERT [54], we use Roberta Tokenizer for tokenization. It adopts byte-level BPE encoding, which can handle the complexity and flexibility of vocabulary. So it is suitable for handling complex inputs such as source code. For GraphCodeBERT [55], in addition to inputting the source code sequence into the model, we also need to parse the source code into AST and extract the sequence of variables. Finally, the dependency relationship between variables is extracted based on variable sequences and AST. Then, we combine word embedding and position embedding to generate the embedding vector of the final input model. Word Embedding converts the semantic information of the word itself into a vector representation. Position Embedding converts the positional information of the word into a vector representation. The position embedding can represent the relative position of the given token in the input sequence. Finally, we concatenate the word embedding and position embedding to generate the input vector for the Transformer encoder block. Such embedding vectors can simultaneously capture the semantics of code tokens and their positions in the input sequence, providing important feature representations for further code analysis and prediction.

3) Training the Second-stage model: We input the embedding vectors into the second-stage model for training. In the second stage, the model performs deep coding and representation learning on these vectors. At the end of the model, we input them into a simple fully connected layer that, by learning appropriate weights and bias parameters, can transform the embedding vectors into a set of probability prediction values $p = [p_1, p_2, ..., p_n]$, which displays the probability that the sample belongs to each class. Finally, we use a *softmax function* to process this set of probabilities so that the sum of probabilities for all classes is equal to 1. In this way, the class with the highest probability can be output as the final prediction $\hat{y}$ in equation 3.

$$\hat{y} = argmax(softmax(p)) \tag{3}$$

E. Two-stage Framework in Inference Phase

In the inference phase, we use the first-stage model to predict the input samples, and then analyze the prediction class and probabilities. If the predicted class is not the ambiguous class, we will directly output the predicted class. However, if the prediction class is the ambiguous class and the sample is located at the decision boundary, we will use the second-stage model to reclassify the sample. Finally, the prediction results of the second-stage model will be used as the final prediction class for the sample.

IV. EXPERIMENTAL DESIGN

In this section, we conduct experiments on two tasks to verify the effectiveness of our method, including emotion recognition in software engineering and vulnerability classification tasks. We will introduce the objectives of each task, the corresponding baseline methods, the datasets we use, and the experimental settings.

A. Dataset and Baseline

We conduct experiments on two different types of software engineering-related classification tasks, involving source code and natural language as input data. This design is intended to verify the effectiveness and generalizability of our method.

1) Software Engineering Emotion Recognition Task: This task is to predict emotion expressed in software engineering-related text. Emotions are pervasive in daily software engineering activities, the detection and analysis of developers' emotions can significantly contribute to maintaining high productivity. Therefore, it is essential to understand developers' emotions, detect negative emotions, and promptly take necessary actions to ensure their sustained productivity. We employ SEntiMoji as our baseline method and use the publicly available dataset from Mia et al.

Baseline: Chen et al. [56] proposed SEntiMoji, a SE-customized emotion classification model. They conclude that

SEntiMoji can significantly outperform existing emotion detection methods in software engineering (e.g., DEVA [57], EMTk [58], MarValous [59]). SEntiMoji is developed based on DeepMoji [60], which is based on Bi-LSTM and Attention, and trained on a small amount of labeled software engineering-related data as well as large-scale emoji-labeled data from both Twitter and GitHub. To adapt to our two-stage framework, we used SEntiMoji for multi-classification, while the original literature used it for binary classification. We replicated the Delegation method [50] and also used it as the baseline method.

Dataset: Mia et al. [61] collected 2,000 instances from pull requests and issues on four popular GitHub repositories and augmented the data using three strategies: Unconstrained, Lexicon, and Polarity. We choose the data augmented by the Polarity strategy because according to the experimental results in [61], it performed the best. Consequently, the augmented dataset for the emotion recognition task in software engineering comprises a total of 15,436 samples.

2) Vulnerability Classification Task: This task is to classify vulnerability code. Software vulnerabilities are common in software systems and can lead to various issues, including deadlocks, data loss, or system failures. By classifying the code that contains vulnerabilities, we can better understand the characteristics and patterns of different types of vulnerabilities, which can provide guidance and support for developers to identify and fix potential vulnerabilities. So, this task is important in the fields of software development and security.

Baseline: Our objective is to classify the vulnerable code, which is different from existing research that primarily focuses on classifying code as vulnerable or non-vulnerable. Therefore, instead of comparing with existing work, we select a widely used BERT [18] model as the baseline for the multi-classification of vulnerability codes. We replicated the Delegation method [50] and also used it as the baseline method.

Dataset: The dataset proposed by Fan et al. [62] is one of the largest vulnerability datasets, consisting of 144,428 instances collected from 348 open-source Github projects between 2002 and 2019, covering 91 different CWEs. Benjamin et al. [63] categorized the samples of the MSR dataset into five types according to the annotated CWE types, i.e., buffer overflow, value error, resource error, input validation error, and privilege escalation. We use the vulnerability code dataset annotated by Benjamin et al. based on MSR for classification. After removing the non-vulnerability code, we obtain a total of 8,379 samples for vulnerability code classification.

B. Model Selection

We use the original model as the first-stage model. For the selection of the second-stage model, based on the recommendation models listed in Section III, we make the following choices for two tasks. For the software engineering emotion recognition task, we select BERT [18] as the second-stage model, which complements the first-stage LSTM model and is well-suited for natural language data. For the vulnerability classification task, considering the input as source code and should be different from the first-stage model, we choose CodeBERT [54] as the second-stage model.

C. Experiment Metrics

To evaluate the effectiveness of our method, we choose popular metrics used to evaluate a classification task: Precision, Recall, and F1-score, which is the average of the prior two. We list the formula used for calculating Precision, Recall, and F1-score below.

Precision is the proportion of true positive observations to the total predicted positive observations.

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

Recall is the proportion of true positive observations to all observations in the actual class.

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

F1-score is a weighted average of Precision and Recall.

$$F1\text{-}score = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \tag{6}$$

Here, TP (True Positive) is the number of samples that the model correctly classifies into the corresponding class; FP (False Positive) is the number of samples incorrectly classified into this class; FN (False Negative) is the number of samples incorrectly classified into other classes.

D. Environments

All the experiments are conducted on a computer containing an 8-core Intel CPU with 64GB physical memory, and a single NVIDIA V100 GPU. The operating system is Ubuntu 18.04.5.

E. Parameter settings

For the software engineering emotion recognition task, we use an initial model with original hyperparameters. The second-stage model is based on the PyTorch implementation of BERT. The padding size is set to 48. The batch size is 16 for training and 32 for validation.

For the vulnerability classification task, the first-stage model is based on the PyTorch implementation of BERT. The padding size is set to 512. The second-stage model is based on the PyTorch implementation of CodeBERT. We use AdamW as the optimizer, with an initial learning rate of 2e-5 and a padding size of 512. The batch size is 16 for both training and validation.

For all models based on BERT implementation, the word embedding size is 768, and we use BertAdam as the optimizer with an initial learning rate of 5e-5, which linearly increases from 0 during a warmup period. For splitting the original dataset, we follow the same splitting strategy as the original literature to ensure that the data used for testing and non-testing parts are consistent with the baseline.

For the Delegation method, we use the parameters set by Ferri et al. [50] in the paper and trained the second stage classifier with 50% low confidence data.

V. Results

Here we discuss the results of our study by placing them in the context of two research questions, RQ1 and RQ2, as discussed later.

A. RQ1:How effective is the two-stage framework in correcting ambiguous classification?

To answer this research question, we conduct experiments on two software engineering classification tasks and compare the two-stage framework with the baseline method. Similar to the evaluation method used in the baseline, we assess the performance using a misclassification rate and three popular classification metrics, i.e., Precision, Recall, and F1-score.

For the software engineering emotion recognition task, we identify two ambiguous class pairs, represented by *(Fear, Anger)* and *(Love, Joy)*. For the vulnerability classification task, we identify one ambiguous class pair, represented by *(Input validation error, Privilege escalation)*.

As shown in Table II and Table III, our two-stage framework improves the performance of the model on ambiguous classes without affecting its predictive performance for other classes. However, the Delegation method does not significantly improve the predictive performance of ambiguous classes and even has side effects. Fig. 3 further demonstrates the effectiveness of our method in reducing misclassification rates of ambiguous classes.

For the software engineering emotion recognition task, our method achieves an F1-score of 56.52% on the class *Anger*, a relative improvement of 5.08% from baseline. Obtains an F1-score of 67.42% on the class *Love*, a relative improvement of 10.2% compared to the baseline. Obtains an F1-score of 61.82% on the class *Joy*, a relative improvement of 2.42% compared to the baseline. We achieve an F1-score of 57.97% on the class *Fear*, which is 19.8% higher than the baseline. Overall, our method achieves an F1-score of 61.75%, a relative improvement of 5.7% compared to the baseline. In terms of misclassification rate, our method reduces the misclassification rate of *(Fear, Anger)* from 36% to 26%, a reduction of 10%, and the misclassification rate of *(Love, Joy)* is reduced from 32% to 18%, a reduction of 14%.

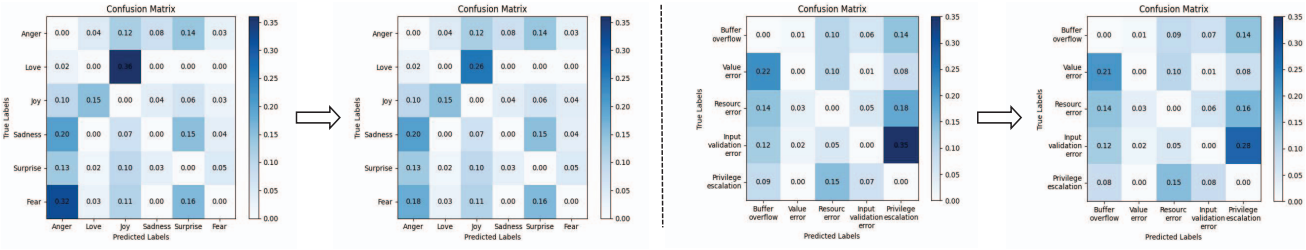
For vulnerability classification tasks, our method achieves an F1-score of 56.35% on the class *Input Validation Error*, a relative improvement of 8.62% compared to the baseline. We achieve an F1-score of 57.07% on the class *Privilege Escalation*, a relative improvement of 2.46% compared to the baseline. Overall, our method achieves an F1-score of 62.52%, a relative improvement of 1.91% compared to the baseline. In terms of misclassification rate, our method reduces the misclassification rate of *(Input validation error, Privilege escalation)* from 35% to 28%, a reduction of 7%.

We also conduct t-tests on these two tasks to demonstrate the statistical significance of the performance improvement brought about by the two-stage framework. Specifically, for the software engineering emotion recognition task, we obtain a p-value of 0.026 ($p<0.05$), indicating a significant difference between the two sets of data. In the case of the vulnerability classification task, we obtain a p-value of 0.018 ($p<0.05$), which similarly indicates a significant difference between the two data sets.

We can observe that it is effective to use the second-stage model to reclassify the samples belonging to the ambiguous classes and located at the decision boundary. The effectiveness of our method may be attributed to the ability of the first-stage model and the second-stage model to capture different aspects or perspectives of the data. The first-stage model may emphasize certain features, while the second-stage model may focus on different aspects and discover additional discriminative features. By combining two different models, their complementary knowledge can be utilized to maximize the classification ability of the two-stage model. However, the effectiveness of the Delegation method is not ideal for two possible reasons. Firstly, this method utilizes the same classifier in both stages, limiting the ability to effectively learn diverse information from the data. As a result, information that the first-stage classifier fails to learn is also difficult to be learned by the second-stage classifier. Secondly, the reduced dataset used to train the second-stage classifier may lead to overfitting, thereby diminishing the performance of the second-stage classifier.

TABLE II
Performance of our two-stage framework and baseline.

Emotion Type	Model	Precision	Recall	F1-score	Relative Improve in F1-score
Anger	SEntiMoji	0.4937	0.5909	0.5379	5.08%↑
	Delegation	0.4500	0.5455	0.4932	
	Two-stage	0.5417	0.5909	**0.5652**	
Love	SEntiMoji	0.6047	0.6190	0.6118	10.20%↑
	Delegation	0.6048	0.6188	0.6108	
	Two-stage	0.6383	0.7143	**0.6742**	
Joy	SEntiMoji	0.5795	0.6296	0.6036	2.42%↑
	Delegation	0.5500	0.5432	0.5466	
	Two-stage	0.6071	0.6296	**0.6182**	
Sadness	SEntiMoji	0.7500	0.5455	0.6316	Non Ambiguous Class
	Delegation	0.6744	0.5273	0.5918	
	Two-stage	0.7500	0.5455	0.6316	
Surprise	SEntiMoji	0.6000	0.6774	0.6364	
	Delegation	0.5375	0.6935	0.6065	
	Two-stage	0.6000	0.6774	0.6364	
Fear	SEntiMoji	0.6250	0.3947	0.4839	19.80%↑
	Delegation	0.6111	0.2895	0.3929	
	Two-stage	0.6452	0.5263	**0.5797**	
Overall	SEntiMoji	0.6088	0.5762	0.5842	5.70%↑
	Delegation	0.5713	0.5363	0.5403	
	Two-stage	0.6304	0.6140	**0.6175**	



(a) (b) (c) (d)

Fig. 3. The effectiveness of two-stage frameworks in reducing misclassification rates.

TABLE III
PERFORMANCE OF OUR TWO-STAGE FRAMEWORK AND BASELINE.

Vulnerability type	Model	Precision	Recall	F1-score	Relative Improve in F1-score
Buffer overflow	BERT	0.7157	0.6857	0.7007	Non Ambiguous Class
	Delegation	0.5077	0.8214	0.6276	
	Two-stage	0.7157	0.6857	0.7007	
Value error	BERT	0.8088	0.5978	0.6875	
	Delegation	0.0000	0.0000	0.0000	
	Two-stage	0.8088	0.5978	0.6875	
Resource error	BERT	0.6105	0.5966	0.6034	
	Delegation	0.5397	0.3864	0.4503	
	Two-stage	0.6105	0.5966	0.6034	
Input validation error	BERT	0.6078	0.4526	0.5188	8.62%↑
	Delegation	0.3708	0.2409	0.2920	
	Two-stage	0.6174	0.5182	**0.5635**	
Privilege escalation	BERT	0.4681	0.6875	0.5570	2.46%↑
	Delegation	0.4746	0.5250	0.4985	
	Two-stage	0.4910	0.6813	**0.5707**	
Overall	BERT	0.6423	0.6040	0.6135	1.91%↑
	Delegation	0.3786	0.3947	0.3737	
	Two-stage	0.6488	0.6159	**0.6252**	

B. RQ2:What are the contributions of different parts to the two-stage framework?

To answer this research question, we design ablation experiments to test the effect of three important parts of our model, i.e., the first-stage model, the second-stage model, and the Boundary Sample Filtering strategy(BSF). The experimental results are shown in Table IV and Table V.

We conduct experiments by using only the first-stage model or only the second-stage model. We also experiment with and without considering BSF, which means that we consider all samples classified into the ambiguous class by the first-stage model, not only samples located at the decision boundary of the ambiguous class. This expands our scope of reclassification. The first two rows show the results of using only the first-stage model and only the second-stage model. The results with (w/) and without(w/o) boundary sample filtering(BSF) strategies are displayed in the next two rows.

As shown in Table IV, for the software engineering emotion recognition task, the two-stage framework achieves higher F1-scores on many ambiguous classes compared to single-stage models, particularly in classes such as *Love*, *Joy*, and *Fear*. Although the F1-score of the two-stage framework on class *Anger* is not higher than that of the second-stage model only, the difference between them is not significant. In the case of (w/o BSF), the F1-score of the most ambiguous class is significantly lower than that of (w/ BSF), including class *Anger*, *Joy*, and *Fear*. While in the case of (w/o BSF), the F1-score of the two-stage framework on the class *Love* is higher than that of (w/ BSF), this is at the cost of the higher misclassification probability of the class *Joy* relative to it. In addition, we are surprised to find that for the class *Fear*, the F1-score using only the first stage model and only the second stage model is 48.39% and 50.00%, but the two-stage framework can improve the F1-score of the class *Fear* to 57.97%. This finding can intuitively illustrate that combining two different models can make full use of knowledge complementarity to maximize the ability of the two-stage model.

As shown in Table V, for the vulnerability classification task, the F1-score of the two-stage framework on ambiguous classes surpasses that of the first-stage only model. In the case of (w/o BSF), the F1-score for the ambiguous classes, including *Input Validation Error* and *Privilege Escalation*, is lower compared to the (w/ BSF) case. Surprisingly, for the class *Input Validation Error*, the F1-score using only the first-stage model and only the second-stage model is only 51.88% and 50.43%, but the two-stage framework can improve the F1-score of the class *Input Validation Error* to 56.35%. This finding can intuitively illustrate that combining two different models can make full use of knowledge complementarity to maximize the ability of the two-stage model.

Combining the ablation experimental results of the two-stage model on two tasks, we find that the two models used in the two-stage framework are able to capture different aspects or perspectives of the data so that the combined effect is better than that of a single use. When removing the BSF, we observe a decrease in F1-score. One possible explanation is that the BSF restricts the second-stage model to focus only on ambiguous class boundary samples. This means that samples confidently classified by the first-stage model are not reclas-

TABLE IV
EMOTION CLASSIFICATION: EFFECTIVENESS OF EACH COMPONENT IN TWO-STAGE FRAMEWORK.

Model	F1-score						
	Anger	Love	Joy	Sadness	Surprise	Fear	Overall
first-stage only	0.5379	0.6118	0.6036	0.6316	0.6364	0.4839	0.5842
second-stage only	**0.5738**	0.6667	0.6154	0.5825	0.6364	0.5000	0.5958
two-stage w/o BSF	0.5147	**0.7083**	0.6076	0.6316	0.6364	0.5070	0.6009
two-stage w/ BSF	0.5652	0.6742	**0.6182**	**0.6316**	**0.6364**	**0.5797**	**0.6175**

TABLE V
VULNERABILITY CLASSIFICATION: EFFECTIVENESS OF EACH COMPONENT IN TWO-STAGE FRAMEWORK.

Model	F1-score					
	Buffer overflow	Value error	Resource error	Input validation error	Privilege escalation	Overall
first-stage only	0.7007	0.6875	0.6034	0.5188	0.5570	0.6135
second-stage only	**0.7070**	0.6415	0.5785	0.5043	**0.6224**	0.6107
two-stage w/o BSF	0.7007	0.6875	0.6034	0.5603	0.5608	0.6225
two-stage w/ BSF	0.7007	**0.6875**	**0.6034**	**0.5635**	0.5707	**0.6252**

sified in the second stage. However, for samples where the first-stage model lacks confidence in classification, the second-stage model takes over to reclassify them. By narrowing the scope of the second-stage model, the probability of incorrectly reclassifying samples correctly classified by the first-stage model is reduced. Overall, the F1-score of the two-stage model outperforms that of the single-stage classification model alone, particularly in most ambiguous classes. Notably, when considering the case with BSF (w/ BSF), the improvement is more pronounced.

VI. DISCUSSION

A. Complementarity of Models

Taking the two model combinations used in Section IV of this paper as an example, discuss the complementarity of the two-stage models. We first train and test these models on the same dataset, and the results are presented through the confusion matrices shown in Fig. 4.

Confusion matrices **(a)** and **(b)** show the classification performance of the LSTM and the BERT on the dataset proposed by Mia et al. [61]. It can be observed that BERT performs better when LSTM tends to misclassify class *Anger* and *Fear*, as well as class *Love* and *Joy*. This indicates that the BERT is able to compensate for the weaknesses of the LSTM in these classes.

Similarly, confusion matrices **(c)** and **(d)** show the classification performance of the BERT and the CodeBERT on the dataset proposed by Benjamin et al. [63]. It can be observed that the CodeBERT outperforms the BERT in cases where the BERT tends to misclassify the class *Input validation error* and *Privilege escalation*. This suggests that CodeBERT is able to compensate for the deficiencies of the BERT in these classes.

In addition, we further analyze the experimental results of the two-stage framework in two tasks. We find that among the samples correctly classified by the first-stage model, only a very small number of samples were misclassified by the second-stage model, accounting for 0 and 0.03%, respectively. This indicates that our two-stage framework can maintain the samples correctly classified by the first-stage model in most cases, and more importantly, correct the samples with incorrect classification by the first-stage model.

Based on the above analysis, the different models used in the two-stage framework have a certain degree of complementarity, which is due to their structural differences and differences in pre-trained knowledge. Although the second-stage model cannot completely correct the errors of the first-stage model, it can largely correct some of the errors, thereby improving the accuracy of the model's classification of ambiguous classes.

B. Threats To Validity

Several limitations may impact the interpretation of our method. We list each of them below.

- A threat to the effectiveness of our study is that we did not use cross-validation to evaluate the performance of the method. This experimental setup may limit our comprehensive evaluation of model robustness. However, in order to make a fair comparison, we followed the same data-splitting strategy as the original literature to ensure that the data used for testing and non-testing parts are consistent with the baseline.
- Another potential threat is that our experiment considered only two combinations of the two-stage framework. However, different tasks and first-stage models can be combined with various second-stage models to form a two-stage framework. To reduce this limitation, we have

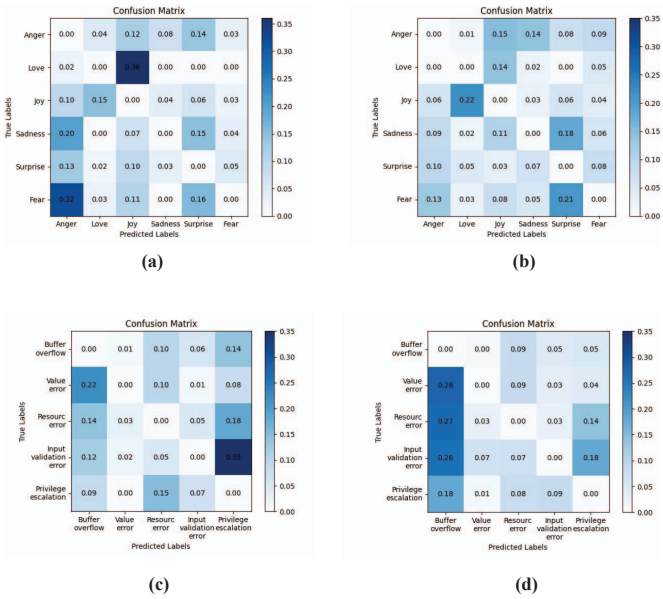


Fig. 4. Performance of different models on the same data.

listed classification model combination recommendations in section III. We leave the optimization of the second-stage model selection strategy and extend the model library for future work.

- Although our method has experimented on two tasks, its basic principles are also applicable to other software engineering-related classification tasks that exhibit ambiguous classification problems. For such scenarios, the framework can be extended by identifying these ambiguous classes and training another model with complementary knowledge, thus establishing a two-stage framework. However, in certain cases where the task does not involve ambiguous classification problems, our method may not yield significant performance improvements.

VII. Related Work

A. Classification Model Testing

An increasing number of works focus on DNN-based classification model testing. Existing research has proposed testing methods based on neuron coverage [38]–[43]. Pei et al. [38] designed and implemented DeepXplore, a system for testing DL, which for the first time introduced neuron coverage as a metric. DeepXplore was able to discover thousands of erroneous behaviors in fifteen state-of-the-art DNNs trained on five real-world datasets. Ma et al. [39] further defined both neuron and hierarchical coverage standards to help measure the testing quality of deep learning models. Sun et al. [40] proposed a DNNs testing and debugging tool called DeepConcolic, which has a high neuron coverage rate and can discover a large number of adversarial instances.

B. Classification Model Repairing

There is a lot of existing research dedicated to correcting bugs in DNN-based classification model. Some research has primarily aimed at discovering models bug patterns and building automated methods for repair [44], [64], [65]. For instance, Zhang et al. [66] studied bug patterns in *Tensorflow* using both GitHub and Stack Overflow. They discussed the new patterns and bug features of *Tensorflow* users writing DNNs applications. They also discussed three new challenges in detecting and locating these bugs. Pham et al. [67] proposed CRADLE, a method focused on detecting and locating bugs in deep learning software libraries. Islam et al. [68] conducted a more comprehensive study by exploring five deep learning libraries, delving into the specific repair patterns, common challenges faced by developers, and the potential introduction of new errors during bug fixing.

Other research has primarily focused on techniques such as data augmentation and retraining to enhance the overall accuracy of classification models [30]–[32]. For instance, Ren et al. [33] proposed FSGMix, a data augmentation-based repair method that maximizes the utilization of limited failure cases to enhance training data. Ma et al. [34] proposed and implemented MODE, an automated neural network debugging tool driven by state difference analysis and input selection. This tool can effectively identify defective neurons and select high-quality training samples to repair model errors.

Additionally, there have been efforts to improve the robustness of models against adversarial instances [23]–[27]. For example, Papernot et al. [28] introduced defensive distillation, a technique that effectively defends against adversarial samples while minimizing changes to the DNNs structure and minimizing the impact on model accuracy. Gu et al. [29] presented the Deep Contractive Network(DCN), a novel end-to-end training process that utilizes layered contraction penalties and smoothness penalties to improve the robustness of deep neural networks against adversarial samples. Existing research focuses on correcting classification errors at the instance level and across the entire dataset of models. In contrast, On the contrary, our research is specifically aimed at reducing ambiguous classification errors in classification models applied to software engineering-related classification tasks, which is a class-level error.

VIII. Conclusion

In this paper, we define ambiguous classification in software engineering classification models and present a two-stage framework to correct ambiguous classification. The experimental results show that our method significantly reduces ambiguous classification errors without sacrificing the overall performance of the model, and improves the F1-score of the ambiguous class. We will expand the second-stage model library and optimize the second-stage model selection strategy for future work.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No.2023RC1001), and NSFC No.62202474.

REFERENCES

[1] S. Kolek, D. A. Nguyen, R. Levie, J. Bruna, and G. Kutyniok, "Cartoon explanations of image classifiers," in *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XII*. Springer, 2022, pp. 443–458.

[2] K. Nakata, Y. Ng, D. Miyashita, A. Maki, Y.-C. Lin, and J. Deguchi, "Revisiting a knn-based image classification system with high-capacity storage," in *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXVII*. Springer, 2022, pp. 457–474.

[3] Y. Liang, Y. Long, Y. Li, and J. Liang, "Selective pseudo-labeling and class-wise discriminative fusion for sound event detection," *arXiv preprint arXiv:2203.02191*, 2022.

[4] G. Datta, T. Etchart, V. Yadav, V. Hedau, P. Natarajan, and S.-F. Chang, "Asd-transformer: Efficient active speaker detection using self and multimodal transformers," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 4568–4572.

[5] C. Lv, J. Xu, and X. Zheng, "Spiking convolutional neural networks for text classification," in *The Eleventh International Conference on Learning Representations*, 2023.

[6] X. Hu, X. Kong, and K. Tu, "A multi-grained self-interpretable symbolic-neural model for single/multi-labeled text classification," *arXiv preprint arXiv:2303.02860*, 2023.

[7] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[8] H. Zhang, L. Yu, X. Xiao, Q. Li, F. Mercaldo, X. Luo, and Q. Liu, "Tfe-gnn: A temporal fusion encoder using graph neural networks for fine-grained encrypted traffic classification," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2066–2075.

[9] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: an image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2365–2376.

[10] M. Fu and C. Tantithamthavorn, "Linevul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[11] C. Baker, L. Deng, S. Chakraborty, and J. Dehlinger, "Automatic multi-class non-functional software requirements classification using neural networks," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, vol. 2. IEEE, 2019, pp. 610–615.

[12] X. Luo, Y. Xue, Z. Xing, and J. Sun, "Prcbert: Prompt learning for requirement classification using bert-based pretrained language models," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[13] E. Bogomolov, V. Kovalenko, Y. Rebryk, A. Bacchelli, and T. Bryksin, "Authorship attribution of source code: A language-agnostic approach and applicability in software engineering," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 932–944.

[14] Z. Li, G. Chen, C. Chen, Y. Zou, and S. Xu, "Ropgen: Towards robust code authorship attribution via automatic coding style transformation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1906–1918.

[15] D. Wang, Y. Yu, S. Li, W. Dong, J. Wang, and L. Qing, "Mulcode: A multi-task learning approach for source code understanding," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 48–59.

[16] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 287–298.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[19] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimarães, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019.

[20] Z. Xu and J. H. Saleh, "Machine learning for reliability engineering and safety applications: Review of current status and future opportunities," *Reliability Engineering & System Safety*, vol. 211, p. 107530, 2021.

[21] M. Hossain and H. Chen, "Application of machine learning on software quality assurance and testing: A chronological survey," *International Journal of Computers and their Applications*, vol. 29, no. 3, pp. 150–157, 2022.

[22] Y. Yang, X. Xia, D. Lo, T. Bi, J. Grundy, and X. Yang, "Predictive models in software engineering: Challenges and opportunities," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–72, 2022.

[23] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, "Adversarial example defense: Ensembles of weak defenses are not strong." in *WOOT*, 2017, pp. 15–15.

[24] F. Tramer, N. Carlini, W. Brendel, and A. Madry, "On adaptive attacks to adversarial example defenses," *Advances in neural information processing systems*, vol. 33, pp. 1633–1645, 2020.

[25] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, "A rotation and a translation suffice: Fooling cnns with simple transformations," Dec 2017.

[26] T. Strauss, M. Hanselmann, A. Junginger, and H. Ulmer, "Ensemble methods as a defense to adversarial perturbations against deep neural networks," *arXiv preprint arXiv:1709.03423*, 2017.

[27] Z. Zhong, Y. Tian, and B. Ray, "Understanding local robustness of deep neural networks under natural variations," in *Fundamental Approaches to Software Engineering: 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 24*. Springer International Publishing, 2021, pp. 313–337.

[28] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 582–597.

[29] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[30] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proceedings of the acm/ieee 42nd international conference on software engineering*, 2020, pp. 1147–1158.

[31] J. Sohn, S. Kang, and S. Yoo, "Search based repair of deep neural networks," *arXiv preprint arXiv:1912.12463*, 2019.

[32] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 177–188.

[33] X. Ren, B. Yu, H. Qi, F. Juefei-Xu, Z. Li, W. Xue, L. Ma, and J. Zhao, "Few-shot guided mix for dnn repairing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 717–721.

[34] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 175–186.

[35] T. G. Dietterich *et al.*, "Ensemble learning," *The handbook of brain theory and neural networks*, vol. 2, no. 1, pp. 110–125, 2002.

[36] R. Kohavi, "Glossary of terms," *Special issue on applications of machine learning and the knowledge discovery process*, vol. 30, no. 271, pp. 127–132, 1998.

[37] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[38] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[39] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 120–131.

[40] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 109–119.

[41] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[42] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 614–618.

[43] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empirical Software Engineering*, vol. 25, pp. 5193–5254, 2020.

[44] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[45] B. Settles, "Active learning literature survey," 2009.

[46] A. J. Joshi, F. Porikli, and N. Papanikolopoulos, "Multi-class active learning for image classification," in *2009 ieee conference on computer vision and pattern recognition*. IEEE, 2009, pp. 2372–2379.

[47] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996.

[48] Y. Freund and R. Schapire, "Experiments with a new boosting algorithm," *International Conference on Machine Learning,International Conference on Machine Learning*, Jul 1996.

[49] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[50] C. Ferri, P. Flach, and J. Hernández-Orallo, "Delegating classifiers," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 37.

[51] D. Kuipers and M. Fabro, "Control systems cyber security: Defense in depth strategies," Idaho National Lab.(INL), Idaho Falls, ID (United States), Tech. Rep., 2006.

[52] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning–based text classification: a comprehensive review," *ACM computing surveys (CSUR)*, vol. 54, no. 3, pp. 1–40, 2021.

[53] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[54] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[55] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[56] Z. Chen, Y. Cao, X. Lu, Q. Mei, and X. Liu, "Sentimoji: an emoji-powered learning approach for sentiment analysis in software engineering," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 841–852.

[57] M. R. Islam and M. F. Zibran, "Deva: sensing emotions in the valence arousal space in software engineering text," in *Proceedings of the 33rd annual ACM symposium on applied computing*, 2018, pp. 1536–1543.

[58] F. Calefato, F. Lanubile, N. Novielli, and L. Quaranta, "Emtk-the emotion mining toolkit," in *2019 IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering (SEmotion)*. IEEE, 2019, pp. 34–37.

[59] M. R. Islam, M. K. Ahmmed, and M. F. Zibran, "Marvalous: Machine learning based detection of emotions in the valence-arousal space in software engineering text," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1786–1793.

[60] B. Felbo, A. Mislove, A. Søgaard, I. Rahwan, and S. Lehmann, "Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm," *arXiv preprint arXiv:1708.00524*, 2017.

[61] M. M. Imran, Y. Jain, P. Chatterjee, and K. Damevski, "Data augmentation for improving emotion recognition in software engineering communication," *arXiv preprint arXiv:2208.05573*, 2022.

[62] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[63] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," *arXiv preprint arXiv:2212.08109*, 2022.

[64] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 561–572.

[65] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "Deepfd: automated fault diagnosis and localization for deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 573–585.

[66] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[67] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[68] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1135–1146.