MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation

Zeming Dong[†], Qiang Hu[‡]*, Yuejun Guo[§], Maxime Cordy[‡],
Mike Papadakis[‡], Zhenya Zhang[†], Yves Le Traon[‡], and Jianjun Zhao[†]
[†]Kyushu University, Japan, dong.zeming.011@s.kyushu-u.ac.jp, {zhang, zhao}@ait.kyushu-u.ac.jp
[‡]University of Luxembourg, Luxembourg, {qiang.hu, maxime.cordy, michail.papadakis, Yves.LeTraon}@uni.lu
[§]Luxembourg Institute of Science and Technology, Luxembourg, yuejun.guo@list.lu

Abstract*—Inspired by the great success of Deep Neural Networks (DNNs) in *natural language processing (NLP)*, DNNs have been increasingly applied in *source code analysis* and attracted significant attention from the software engineering community. Due to its data-driven nature, a DNN model requires massive and high-quality labeled training data to achieve expert-level performance. Collecting such data is often not hard, but the labeling process is notoriously laborious. The task of DNN-based code analysis even worsens the situation because source code labeling also demands sophisticated expertise. Data augmentation has been a popular approach to supplement training data in domains such as computer vision and NLP. However, existing data augmentation approaches in code analysis adopt simple methods, such as data transformation and adversarial example generation, thus bringing limited performance superiority. In this paper, we propose a data augmentation approach MixCode that aims to effectively supplement valid training data, inspired by the recent advance named *Mixup* in computer vision. Specifically, we first utilize multiple code refactoring methods to generate transformed code that holds consistent labels with the original data. Then, we adapt the Mixup technique to mix the original code with the transformed code to augment the training data. We evaluate MixCode on two programming languages (Java and Python), two code tasks (problem classification and bug detection), four benchmark datasets (*JAVA250*, *Python800*, *CodRep1*, and *Refactory*), and seven model architectures (including two pre-trained models *CodeBERT* and *GraphCodeBERT*). Experimental results demonstrate that MixCode outperforms the baseline data augmentation approach by up to 6.24% in accuracy and 26.06% in robustness.

***Index Terms*—Data augmentation, Mixup, Source code analysis**

I. Introduction

Due to its remarkable performance, deep learning (DL) has gained widespread adoption in different application domains, such as face recognition [1], language translation [2], video games [3], and autonomous driving [3]. More recently, researchers from the software engineering community have attempted to use DL techniques to automate multiple downstream code tasks, e.g., code search [4], problem classification [5], and bug detection [6]. Relevant studies [7], [8] reveal that DL benefits source code analysis.

As the key pillar of DL systems, deep neural networks (DNNs) automatically gain knowledge from training data and make inferences for unseen data after deployment. Generally, two important factors could affect the performance of the trained DNNs, namely, the *model architecture* and the *training data.* In the context of code analysis, for the former factor, a common practice of building proper model architectures of DNNs is to directly apply natural language processing (NLP) models to source code. For example, Feng et al. [7] have modified BERT to create *CodeBERT* that solves downstream tasks effectively. For the latter factor, though adequate labeled training data are necessary for the training process, producing high-quality labeled source code data is not yet sufficiently investigated. The main challenge is that data labeling requires not only extensive human efforts but also sophisticated domain knowledge. According to to [9], labeling code from only four libraries can take up to 600 man-hours. In a nutshell, data preparation is indispensable but challenging for developing desirable models, and therefore in this paper, we take a specific focus on this important issue.

Data augmentation is a technique that tackles the aforementioned data labeling issue, which produces additional training data by modifying existing data rather than human efforts. Generally, the new data sample is semantically consistent with the source data, i.e., they share the same functionalities and labels. In this way, the model can learn more information and gain better generalization, compared to the approach that relies only on the original training data. In computer vision and NLP tasks, data augmentation has been widely used and well-studied [10]–[12] for model training. For example, in computer vision tasks, many image transformation methods (e.g., image rotation, shear) are designed to mimic different real-world situations that the model could face after deployment. In traditional NLP tasks, the typical augmentation is to perform synonym substitution, which is also beneficial to cover more context that might occur in the real world.

Although data augmentation has proved to be effective in fields such as CV and NLP, the investigation of its application in code analysis still remains at an early stage. Researchers have borrowed ideas from other fields and proposed several data augmentation approaches for code analysis [13]–[17]; usually, these techniques generate more transformed or adversarial data simply via methods such as code refactoring. However, existing studies [18] already show that these simple strategies have limited effects. For example, Bielik et al. [19] show that adversarial training, by simply adding adversarial

*Qiang Hu is the corresponding author.

data in the training set, is not helpful in improving the generalization property of DNN models. Therefore, it still remains an open problem to design data augmentation approaches that can effectively enhance DNN training for code analysis.

In this paper, for source code classification tasks, we propose a novel data augmentation framework named MIXCODE that aims to enhance the DNN model training process. Roughly speaking, MIXCODE follows a similar paradigm to *Mixup* [20] but adapts the technique in order to handle the specific data type of source code. Mixup is a well-known data augmentation technique originally proposed for image classification, which linearly mixes training data, including their labels, to increase the data volume. In our case, MIXCODE consists of two steps: first, it generates transformed data by different code refactoring methods, and then, it linearly mixes the original code and transformed code to generate new data. Specifically, we study 18 types of existing code refactoring methods, such as argument renaming and statement enhancement. More details are in Section III-B.

We conduct experiments to evaluate the effectiveness of MIXCODE on two programming languages (Java and Python), two widely-studied code learning tasks (problem classification and bug detection), and seven model architectures. Based on that, we answer three research questions as follows:

RQ1: How effective is MIXCODE for enhancing the accuracy and robustness of DNNs? We compare MIXCODE to the standard training (without data augmentation) and the existing simple code data augmentation approach, which relies on transformed or adversarial data only. The results show that MIXCODE outperforms the baselines by up to 6.24% in accuracy and 26.06% in robustness. Here, accuracy is the basic metric that measures the effectiveness of the trained DNN models. Moreover, robustness [21] reflects the generalization ability of the trained model to handle unseen data, which is also an important metric for the deployment of DNN models in practice [22].

RQ2: How do different Mixup strategies affect the effectiveness of MIXCODE? We study the effectiveness of MIXCODE under different settings of Mixup. First, we study the effect of using different data mixture strategies, which involve 1) mixing only original code, 2) mixing original code and transformed code, and 3) mixing only transformed code. Moreover, we also study the effect of different hyperparameters in MIXCODE. Our evaluation demonstrates that using the 2) strategy, namely, mixing original code and transformed data, in Mixup can achieve the best performance, and we also make the suggestion on the use of the most suitable hyperparameters of MIXCODE.

RQ3: How does the refactoring method affect the effectiveness of MIXCODE? To investigate the impact of the code refactoring methods on MIXCODE, we evaluate MIXCODE using different refactoring methods individually. We find that there is a trade-off between the original test accuracy and robustness when choosing different refactoring methods, i.e., using the refactoring methods that lead to higher accuracy could harm the model's robustness.

In summary, the main contributions of this paper are:

- We propose MIXCODE, the first Mixup-based data augmentation framework for source code analysis. Experimental results demonstrate that MIXCODE outperforms the baseline data augmentation approach by up to 6.24% in accuracy and 26.06% in robustness. The implementation of MIXCODE are available online.[1]
- We empirically demonstrate that simply mixing the original code is not the best strategy in MIXCODE. In addition, MIXCODE using original code and transformed code can achieve 9.23% performance superiority in accuracy.
- We empirically show that selection of refactoring methods is also an important factor affecting the performance of MIXCODE.

II. PRELIMINARIES

We briefly introduce the preliminaries of this work from the perspectives of DNNs for source code analysis, DNN model training methods, and Mixup for data augmentation.

A. DNNs for Source Code Analysis

DNNs have been widely used in NLP and achieved great success. Similar to the natural language text, source code also consists of discrete symbols that can be processed as sequential or structural data fed into DNN models. Thus, researchers have tried to employ DNNs to help programmers process and understand source code in recent years. The impressive performance of DNNs has been demonstrated in multiple important code-related tasks, such as automated program repair [23]–[30], automated program synthesis [31], [32], and automated code comments generation [33].

To unlock the potential of DNNs for code-related tasks, properly representing snippets of code that are fed into DNNs is necessary. Code representation, which transfers the raw code to machine-readable data, plays an important role in source code analysis. Existing representation techniques can be roughly divided into two categories, namely, *sequence representation* [34] and *graph representation* [8], [35], [36]. Sequence representation converts the code into a sequence of tokens. The input features of classical neural networks in sequence representation learning are typically embedded or features that live in Euclidean space. In this way, the original source code is processed to multiple tokens, e.g., from "`def func(a, b)`" to "`[def, func, (, a, b, ),]`". Sequence representation is useful for learning the semantic information of the source code because it remains the context of the source code. On the other side, graph representation builds structural data. In source code, the structure information can be presented by the *abstract syntax tree (AST)* and different code flows (control follow, data flow). By learning these structural data, the model can perceive functional information of code. Recently, more researchers have focused on the field that

[1]https://github.com/zemingd/Mixup4Code

applies graph representation to source code analysis based on different variants of *graph neural networks (GNNs)*. In our study, we consider both categories of code representation to evaluate MIXCODE.

B. DNN Model Training Methods

DNN training consists in, given a set of training data, searching for the best parameters (e.g., weights, biases) that enable the model to fit the training data. Here, we introduce the standard training process and the basic data augmentation framework for the source code model. Algorithm 1 presents the pseudocode of these two training strategies. In the standard manner of training, all the training data are fed into several epochs of training (See Lines 1-3 in Algorithm 1).

Algorithm 1 Existing model training strategies

```
Require: $M$: initialized DNN model
Require: $X, Y$: original training data and labels
Require: $R = \{r\}$ : a set of data transformation methods
Ensure: $M$: trained model
Standard training (without augmentation)
 1: for $run \in \{0, \ldots, \#epochs\}$ do
 2:     $M$.Fit $(X, Y)$
 3: return $M$
Basic augmentation
 4: for $run \in \{0, \ldots, \#epochs\}$ do
 5:     $X_{ref} \leftarrow \phi$
 6:     for $x \in X$ do
 7:         $r \leftarrow$ RandomSelection $(R)$
 8:         $X_{ref} \leftarrow X_{ref} \cup r(x)$
 9:     $M$.Fit $(X_{ref}, Y)$
10: return $M$
```

However, since the prepared training data can only represent a limited part of data distribution, the training data volume has been a bottleneck that prevents DNN models from achieving high performance [37]. Data augmentation is proposed to automatically increase the volume of the training set, and thus enhance the quality of training. The basic idea of data augmentation is to generate new data from the existing training data by well-designed data transformation methods. Generally, such data transformation methods modify the data and do not change their semantic information; for example, in image data processing, commonly-used methods include random rotating, padding, and adding brightness [10]. Line 4-10 in Algorithm 1 shows the process of training with data augmentation. Specifically, in each epoch, the DNN is trained by using a transformed version of the data generated by randomly selected data transformation methods.

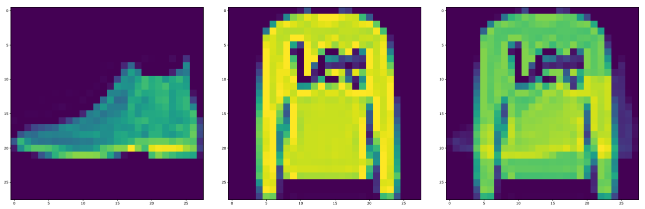
C. Mixup: A Data Augmentation Approach in Image Classification Tasks

Mixup [20] is an effective data augmentation technique proposed for image classification tasks. Mixup contains two steps: first, it randomly selects two data samples from the training data; then, it mixes both the data features and the labels of the selected data to generate a new sample as the training data. In addition to image classification, recently, researchers have achieved great success in applying Mixup to text classification [38].

Technically, Mixup is shown as follows: given a pair of samples (x^i,y^i) and (x^j,y^j), where x represents the input feature and its corresponding output label y is donated with one-hot encoding, Mixup produces new data pairs ($x^{ij}_{mix}, y^{ij}_{mix}$):

$$\begin{aligned} x^{ij}_{mix} &= \lambda x^i + (1-\lambda)x^j \\ y^{ij}_{mix} &= \lambda y^i + (1-\lambda)y^j \end{aligned} \tag{1}$$

where λ is a mixing policy for the input sample pair, which is sampled from a *Beta* distribution with a shape parameter α ($\lambda \sim$ *Beta*(α,α)). Figure 1 depicts an example of an image generated by Mixup. By mixing two images into one, a model can gain knowledge from both sides.



(a) original image 1 (b) original image 2 (c) mixed image

Fig. 1. An example of Mixup for image data. The mixed image is calculated using Eq. (1). $\lambda = 0.2$.

III. MIXCODE—THE PROPOSED APPROACH

A. Methodology of MIXCODE

Inspired by the great success of Mixup and its variants in image classification tasks, we propose MIXCODE, a simple yet effective data augmentation framework for source code classification tasks. Algorithm 2 presents the whole process of MIXCODE. Essentially, Algorithm 2 is different from the existing approaches in Algorithm 1 in the way it augments the training data in each training epoch. Figure 2 presents an overview of MIXCODE in one epoch. Concretely, this process consists of the following three phases:

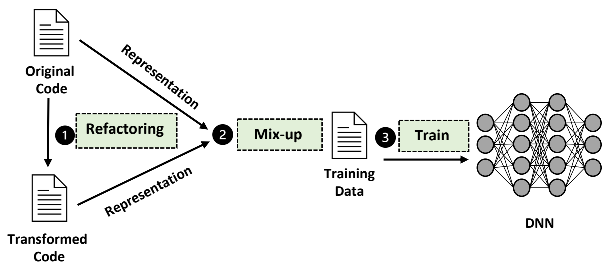
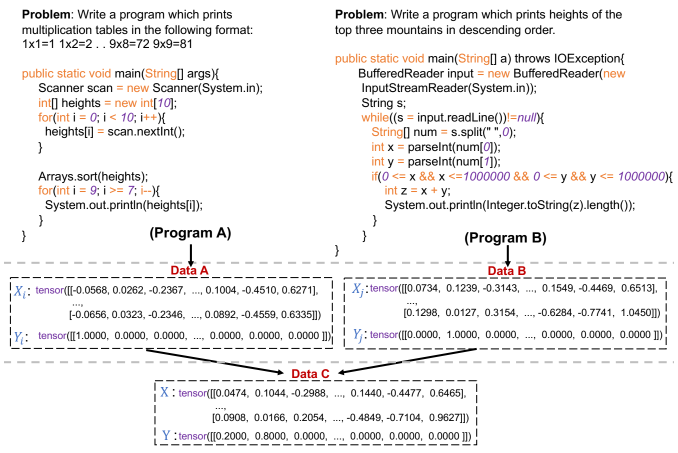


Fig. 2. Workflow of MIXCODE within one training epoch.

Algorithm 2 MIXCODE

```
Require: $M$: initialized DNN model
Require: $T$: code representation technique
Require: $X, Y$: original training data and labels
Require: $R = \{r\}$ : a set of refactoring methods
Require: $\alpha$ : Mixup weight
Ensure: $M$: trained model
 1: for $run \leftarrow \{0, \ldots, \#epochs\}$ do
 2:     $X_{ref}, X_{mix}, Y_{mix} \leftarrow \phi, \phi, \phi$          ▷ initialization
 3:     for $x \in X$ do
 4:         $r \leftarrow$ RandomSelection $(R)$
 5:         $X_{ref} \leftarrow X_{ref} \cup r(x)$          ▷ code refactoring
 6:     $X_s, Y_s \leftarrow$ Shuffle $(X, Y)$     ▷ Shuffle the training set
 7:     for $(x_s, y_s, x_{ref}, y) \in (X_s, Y_s, X_{ref}, Y)$ do
 8:         $v_x \leftarrow T(x_s)$          ▷ code representation
 9:         $v_{ref} \leftarrow T(x_{ref})$          ▷ code representation
10:         $\lambda \leftarrow Beta(\alpha)$          ▷ hyperparameter
11:         $x_{mix} \leftarrow \lambda v_x + (1 - \lambda) v_{ref}$     ▷ data generation
12:         $y_{mix} \leftarrow \lambda y_s + (1 - \lambda) y$     ▷ label generation
13:         $X_{mix} \leftarrow X_{mix} \cup x_{mix}$
14:         $Y_{mix} \leftarrow Y_{mix} \cup y_{mix}$
15:     $M$.Fit $(X_{mix}, Y_{mix})$   ▷ training with augmented data
16: return $M$
```

1) MIXCODE loads the raw data, which consists of the *original code*, from the original training set (X, Y). Then, for each data, it randomly selects one refactoring method (Line 4) and applies it to the original code to obtain the *transformed code* (Line 5). Code refactoring is a technique that restructures the code without changing its semantic behaviors [39], [40]. The current version of MIXCODE supports 18 different refactoring methods, which we elaborate on later in Section III-B. As a result, all the training epochs have different sets of transformed code generated by different refactoring methods.
2) MIXCODE randomly chooses one data from the original code and one data from the transformed code, respectively (Line 8 and 9), and then mixes these two data following Eq. (1), shown as Lines 7-14 in Algorithm 2. Here, v_x and v_{ref}, corresponding to x^i and x^j in Eq. (1), are data under proper code representation. Generally, the data format is a sequence of token values. Moreover, y_s and y in Line 12, corresponding to y^i and y^j, are the one-hot values of labels, which are the same as the original Mixup approach in Section II-C. In this way, we produce new data (x_{mix}, y_{mix}) in a similar way as the original Mixup does, and add them into the training set (X_{mix}, Y_{mix}).
3) Finally, the mixed data set (X_{mix}, Y_{mix}) is used as the training data for this epoch. Note that, although evaluated on image data in [20], the Mixup technique is not limited to the continuous representation space. The core of Mixup is to combine sample-target pairs to increase the training data size. In addition, it has been proven to be applicable to the discrete representation space for NLP tasks [41]. Similarly, MIXCODE supports both Integer-type and Float-type input data by simply setting the input type of embedding layers as float (e.g., `x = torch.tensor(dataset, dtype=torch.float)`).


Problem: Write a program which prints multiplication tables in the following format: 1x1=1 1x2=2 . . 9x8=72 9x9=81

```
public static void main(String[] args){
    Scanner scan = new Scanner(System.in);
    int[] heights = new int[10];
    for(int i = 0; i < 10; i++){
      heights[i] = scan.nextInt();
    }

    Arrays.sort(heights);
    for(int i = 9; i >= 7; i--){
      System.out.println(heights[i]);
    }
}
```

Problem: Write a program which prints heights of the top three mountains in descending order.

```
public static void main(String[] a) throws IOException{
    BufferedReader input = new BufferedReader(new
     InputStreamReader(System.in));
     String s;
     while((s = input.readLine())!=null){
       String[] num = s.split(" ",0);
       int x = parseInt(num[0]);
       int y = parseInt(num[1]);
       if(0 <= x && x <=1000000 && 0 <= y && y <= 1000000){
          int z = x + y;
          System.out.println(Integer.toString(z).length());
       }
     }
}
```

```
Xi: tensor([[-0.0568, 0.0262, -0.2367,  ..., 0.1004, -0.4510, 0.6271],
             ...,
            [-0.0656, 0.0323, -0.2346,  ..., 0.0892, -0.4559, 0.6335]])
Yi:  tensor([[1.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000 ]])

Xj: tensor([[0.0734,  0.1239, -0.3143,  ...,  0.1549, -0.4469,  0.6513],
             ...,
            [0.1298,  0.0127,  0.3154,  ..., -0.6284, -0.7741,  1.0450]])
Yj: tensor([[0.0000,  1.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000 ]])

X : tensor([[0.0474,  0.1044, -0.2988,  ...,  0.1440, -0.4477,  0.6465],
             ...,
            [0.0908,  0.0166,  0.2054,  ..., -0.4849, -0.7104,  0.9627]])
Y : tensor([[0.2000,  0.8000,  0.0000,  ...,  0.0000,  0.0000,  0.0000 ]])
```

Fig. 3. An example of two programs mixed by MIXCODE.

Influence factors. In phase 2, we notice that two factors could affect the performance of MIXCODE, namely, the candidate data pairs (namely, Line 7 in Algorithm 2) used for Mixup and the hyperparameter α. For the candidate data, we have three optional combinations, 1) mixing the two original codes, 2) mixing the original code and transformed data, and 3) mixing two transformed codes. On the other side, the hyperparameter α controls the percentage (λ) of code content used from two parts for Mixup. As mentioned in Section II, λ is a value with the [0, 1] range and is sampled from a Beta distribution [42] parameterized by α, and $\alpha = 0.2$ is the recommended setting for image classification tasks in the original work of Mixup. In our evaluation, we study the settings of α from 0.05 to 0.5 to try to find a suitable setting for source code classification.

B. Refactoring Methods

As we introduced, code refactoring is a technique that restructures the code while keeping its semantic behaviors [39], [40]. The original purpose of code refactoring is to improve readability and reduce code complexity. Thus, programmers can clean up complicated code and reduce technical debt. Meanwhile, refactoring makes it easier for developers to maintain and add new features to the clean code, which is an important step in software maintenance. Typically, refactoring makes a small change in source code that preserves the behavior of the program based on a series of standardized micro-modifies. Several refactoring methods have been proposed and studied, such as replacing a variable, modifying (including adding or simplifying conditional expressions and method calls), moving features between objects, and organizing data.

In the first step of MIXCODE, in addition to the original code, we utilize multiple code refactoring methods to generate more diverse code as the candidate training data. MIXCODE supports 18 types of refactoring methods from

TABLE I
DESCRIPTION AND EXAMPLES OF 18 TYPES OF REFACTORING METHODS.

No.	Refactoring method	Functionality	Example
1	API renaming	Rename an API by a synonym of its name. Only for token-based code learning tasks.	$numpy.add\,() \rightarrow numpy.delete\,()$
2	Arguments adding	Add an unused argument to a function definition.	$def\ func\,(a, b) \rightarrow def\ func\,(a, b, c)$
3	Arguments renaming	Rename an augment by a synonym of its name.	$def\ func\,(number) \rightarrow def\ func\,(size)$
4	Dead for adding	Add an unreachable for loop at a randomly selected location.	add: $for\ i\ in\ range\,(0) : print\,(0)$
5	Dead if adding	Add an unreachable if statement at a randomly selected location in the code.	add: $if\,(1 == 0) : print\,(0)$
6	Dead if else adding	Add an unreachable if-else statement at a randomly selected location in the code.	add: $print\,(0)\,if\ \ (1\ \ ==\ \ 0)\ \ else\ print\,(1)$
7	Dead switch adding	Add an unreachable switch statement at a randomly selected location.	$int\ a = 0;\ switch\ \ (a)\ case\ 1$: $System.out.println\,(\text{“}pass''); break;$ $default: System.out.println\,(\text{“}pass'');$
8	Dead while adding	Add an unreachable while loop at a randomly selected location.	add: $while\,(1 == 0)$: $print\,(0)$
9	Duplication	Duplicate a randomly selected assignment and insert it to its next line.	$a\ =\ 1 \rightarrow a\ =\ 1;\ a\ =\ 1$
10	Filed enhancement	Enhance the rigor of the code by checking if the input of each argument is None.	$def\ \ (a): \rightarrow def\ \ (a):$ $if\ \ a == None: print\,(\text{“}please\ check\ your\ input.'')$
11	For loop enhancement	Enhance the for loop conditions by complementing the lower and upper bound.	$for\ i\ in\ range\,(10) \rightarrow for\ i\ in\ range\,(0,\ 10)$
12	If enhancement	Change an if condition to an equivalent logic.	$if\ True: \rightarrow if\,(0 == 0)$
13	Local variable adding	Add an unused local variable.	add: $a = 1$
14	Local variable renaming	Rename a local variable by a synonym of its name and recursively update all related variables.	$number\ \ = 1 \rightarrow size\ \ = 1$
15	Method name renaming	Rename a method by a synonym of its name.	$def\ count\,(a) \rightarrow def\ compute\,(a)$
16	Plus zero	Select an numerical assignment of mathematical calculation and plus zero to its value.	$a = 1 \rightarrow a = 1 + 0$
17	Print adding	Add a print line at a randomly selected location.	add: $print\,(1)$
18	Return optimal	Change the return content to a variant with the same effect.	$return\ 1 \rightarrow return\ 0\ if\,(1 == 0)\ else\ 1$

the literature [14], [43]. The functionality of each method and a corresponding example are listed in Table I. Note that some transformations may alter code semantics, but not in a way that can affect model decisions, i.e., all transformations are label-preserving. Existing code learning models mainly use two types of code representation: token sequence and abstract syntax tree (AST) [44]. All the MixCode-supported refactoring methods except API Renaming can be applied to these two prepossessing formats, and therefore, MixCode is equipped with strong flexibility.

C. A Case Study

To better understand how MixCode works, we use an example to show its workflow, as depicted in Figure 3. We first transform the source code (Program A and B from the JAVA250 dataset with label 0 and label 1, respectively) into vectors through CodeBERT [7] and the labels into one-hot vectors (see Data A and Data B in Figure 3). X is the value of each token, and Y is the one-hot format label (250 classes in total). Next, we linearly mix the code and label vectors, respectively, of these two programs as the final input to train the model (See Data C in Figure 3). In this example, we set λ in Eq. (1) as 0.2 and perform the input mixing.

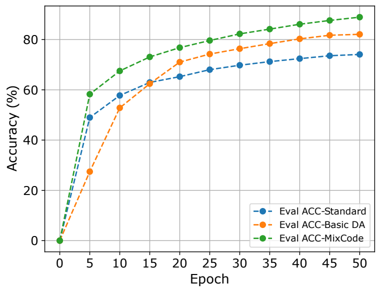


Fig. 4. Test accuracy of models trained by using different methods (dataset: Java250, model: BagofToken).

TABLE II
DETAILS OF DATASETS AND DNNS. #TRAINING, #ORI TEST, AND #ROBUST TEST REPRESENT THE NUMBER OF TRAINING DATA, ORIGINAL TEST DATA, AND TEST DATA FOR GENERALIZATION EVALUATION, RESPECTIVELY.

Dataset	Language	Task	#Training	#Ori Test	#Robust Test	Model
JAVA250	Java	Problem classification	48000	15000	75000	BagofToken SeqofToken CodeBERT GraphCodeBERT
Python800	Python	Problem classification	153600	48000	240000	
CodRep1	Java	Bug detection	6944	772	7716	GCN GAT GGNN CodeBERT GraphCodeBERT
Refactory	Python	Bug detection	3380	423	4225	

Then, we use one example to show the training process of using different training methods, as shown in Figure 4. We can see that MixCode leads to faster model convergence than the other two methods, which draws a similar conclusion to the usage of Mixup (and its variants) in other fields [45]–[47].

IV. Experimental Setup

To evaluate the effectiveness of MixCode, we conduct experiments on two popular programming languages (Java and Python), two important downstream tasks (problem classification and bug detection), and seven DNN model architectures, including two pre-trained model CodeBERT and GraphCodeBERT. Table II presents the details of the datasets and models used in the evaluation.

Task and Dataset. MixCode is suitable for all code classification problems. We select two widely studied tasks, problem classification and bug detection, in our evaluation to demonstrate the effectiveness of MixCode. *Problem classification* is a basic code learning task for collecting the target code from code pools with massive source code (e.g., GitHub). Given a number of problem descriptions and the candidate source code, the model will predict the problem that this code is trying to solve. For this task, our experiments use two recently released datasets, JAVA250 and Python800 [5]. JAVA250 is used for Java program classification with 250 classification problems, and each problem includes 300 Java programs. It also provides two different program representations, which are BagofToken and SeqofToken. Python800 is a dataset for Python program classification, including 800 classification problems with 300 Python programs in each problem. BagofToken and SeqofToken are also available in Python800. *Bug detection* tries to identify whether one code has bugs or not. Generally, bug detection can be seen as a binary classification problem. Preparing bug detection datasets is difficult since it requires a pair of codes with and without bugs. Crawling two versions of code before and after commits from GitHub is the common way to collect such pairs. However, human effort is required to check if the commit is to solve a bug manually. For this task, we use two open datasets, Refactory [48] and CodRep1 [49] in our study. Refactory includes 2,242 correct and 1,783 buggy Python programs written by real-world undergraduate students. CodRep1 provides 3,858 program pairs (buggy program and its fixed version) from real bug fixes. We use each dataset's originally provided training data for the model training.

Model. To avoid the model-depended issue, we build at least four model architectures for each dataset. In problem classification tasks, we follow the recommendation of [5] and build a bag of token-based FNN (BagofToken) and a sequence of token-based CNN (SeqofToken) for each dataset. FNN is a basic DNN type with only dense layers, while CNN is the advanced model type with convolutional layers and achieves great success in image classification tasks. To be simplified, in the remaining part, we use SeqofToken to represent the sequence of token-based CNN, which represents the code with its order of tokens, and BagofToken to represent the bag of token-based FNN, which represents a code sample with the relative frequencies of operator and keyword token occurrences, respectively. In the bug detection tasks, we follow the work [9] and build three types of GNN models, Gated Graph Sequence Neural Networks (GGNNs) [50], Graph Convolutional Networks (GCNs) [51], and Graph Attention Networks (GATs) [52]. GGNN is a graph neural network that updates the new hidden state with Gated Recurrent Unit (GRU). GCN is a variant of convolution neural networks that operate on graph-structured data. GAT mainly applies the attention mechanism to graph message passing. A code graph is an inter-procedural graph of program instructions where edges represent data flow and control flow information. In graph-based code models, we mix the graph data vectors of code representation, which are produced by the GNN using the node and edge embeddings, not code directly. Besides, we also include CodeBERT [7] and GraphCodeBERT [35] in both of the code tasks, as they have demonstrated state-of-the-art performance in a variety of code processing tasks [53], [54]. CodeBERT is a bimodal pre-trained model that takes as input natural language text and code data and produces the general representations of the text and code. GraphCodeBERT is a pre-trained model that represents the code using data-flow information to consider the semantic-level structure of code.

Baseline. We compare MIXCODE with two baselines. The first one is the basic data augmentation method, in which we train the model while conducting random code refactoring on the training data at each training epoch. This data augmentation method is used in existing works [14], [15]. The second baseline is the standard training process without any data augmentation.

Evaluation measures. For the model testing, we evaluate the test accuracy and robustness of DNNs. The test accuracy is the percentage of correctly classified data over the entire test data. For the robustness evaluation, we first generate new test sets by the refactoring methods from the original test set, then calculate the percentage of correctly classified data from this new set. The robustness reflects the generalization ability of the trained model, i.e., the model performance when facing more diverse unseen data.

Implementation and Environments. The pure Python language implements the core code of MIXCODE with only the Numpy package. That means MIXCODE can be easily reused in any DL framework for other classification tasks. We provide two versions of code refactoring methods that support both Java and Python languages. The models of the classification tasks are built using TensorFlow2.3 and Keras2.4.3 frameworks, and the models of the bug detection tasks are built using PyTorch1.6.0. For the SeqOfToken, BagOfToken, CodeBERT, and GraphCodeBERT experiments, we set the training epoch configuration to 50 and the GNNs experiment to 100. We conduct our experiments on an NVIDIA Tesla V100 16G SXM2 GPU. We train each model 5 times to reduce the effect of randomness and report the average results with standard deviation.

V. RESULTS ANALYSIS

A. *RQ1: Effectiveness of* MIXCODE

The upper component in Table III presents the test accuracy of trained models on original test data. The first conclusion to be drawn from the result is that MIXCODE almost always outperforms the two baselines regardless of datasets and models, showing that data augmentation is useful to improve the model performance compared to standard training. However, simply adding data into the training set only slightly improves the basic strategy, especially in bug detection tasks (CodRep1 and Refactory). For example, in Refactory with GAT, the basic data augmentation only improves the accuracy by up to 0.29%. This phenomenon is similar to the findings in [15], which reveal that traditional adversarial training (simply adding adversarial examples to the training data) is not sufficient to improve the robustness of source code models. By contrast, MIXCODE has a significant improvement (up to 5.69%) compared to the basic data augmentation. In bug detection (CodRep1 and Refactory), where the basic data augmentation is not helpful,

TABLE III
EFFECTIVENESS OF MIXCODE CONCERNING TEST ACCURACY AND ROBUSTNESS (AVERAGE± STANDARD DEVIATION, %) ON ORIGINAL TEST DATA AND TRANSFORMED TEST DATA. STANDARD: STANDARD TRAINING WITHOUT DATA AUGMENTATION. BASIC: BASIC DATA AUGMENTATION. THE VALUE WITH A GRAY BACKGROUND INDICATES THE BEST RESULT, AND THE VALUE HIGHLIGHTED IN RED SHOWS THE ACCURACY DIFFERENCE BETWEEN THE BEST AND THE SECOND-BEST. THE HIGHER, THE BETTER.

Dataset	DNN	Standard	Basic	MIXCODE
		Test Accuracy		
JAVA250	**BagofToken**	71.66±0.03	76.63±0.08	82.32±0.07 (5.69↑)
	SeqofToken	86.57±0.06	93.92±0.15	94.52±0.23 (0.60↑)
	CodeBERT	96.37±0.02	96.46±0.07	96.98±0.09 (0.52↑)
	GraphCodeBERT	96.48±0.03	96.51±0.06	97.16±0.12 (0.65↑)
CodRep1	**GGNN**	62.69±0.32	63.07±0.35	65.42±0.35 (2.35↑)
	GCN	61.38±0.44	62.68±0.56	64.18±0.44 (1.50↑)
	GAT	61.27±0.26	62.07±0.33	64.09±0.36 (2.02↑)
	CodeBERT	69.12±0.03	71.22±0.02	72.96±0.05 (1.74↑)
	GraphCodeBERT	70.05±0.11	71.35±0.09	73.34±0.13 (1.99↑)
Python800	**BagofToken**	67.31±0.05	67.46±0.08	68.27±0.08 (0.81↑)
	SeqofToken	82.65±0.32	83.26±0.41	85.00±0.20 (1.74↑)
	CodeBERT	96.05±0.02	96.16±0.01	96.79±0.06 (0.63↑)
	GraphCodeBERT	96.27±0.05	96.29±0.03	97.09±0.08 (0.80↑)
Refactory	**GGNN**	81.96±0.22	81.98±0.23	88.22±0.18 (6.24↑)
	GCN	80.12±0.25	80.23±0.37	84.16±0.25 (3.93↑)
	GAT	79.76±0.19	80.05±0.28	83.38±0.31 (3.33↑)
	CodeBERT	95.88±0.42	96.09±0.33	97.57±0.37 (1.48↑)
	GraphCodeBERT	96.81±0.17	96.92±0.11	98.16±0.21 (1.24↑)
		Robustness		
JAVA250	**BagofToken**	40.21±0.09	52.11±0.06	78.17±0.05 (26.06↑)
	SeqofToken	57.83±0.01	84.94±0.02	85.60±0.01 (0.66↑)
	CodeBERT	89.71±0.09	92.19±0.05	93.17±0.11 (0.98↑)
	GraphCodeBERT	90.56±0.03	92.78±0.07	94.07±0.16 (1.29↑)
CodRep1	**GGNN**	38.84±0.02	50.76±0.03	55.01±0.01 (4.25↑)
	GCN	38.40±0.01	49.97±0.02	54.03±0.03 (4.06↑)
	GAT	38.07±0.02	49.46±0.04	53.81±0.03 (4.35↑)
	CodeBERT	61.11±0.04	62.28±0.03	66.15±0.09 (3.87↑)
	GraphCodeBERT	61.03±0.11	61.86±0.04	65.91±0.13 (4.05↑)
Python800	**BagofToken**	38.76±0.02	39.04±0.02	63.65±0.02 (24.61↑)
	SeqofToken	58.18±0.03	80.81±0.02	82.94±0.02 (2.13↑)
	CodeBERT	89.64±0.04	89.97±0.01	92.16±0.08 (2.19↑)
	GraphCodeBERT	91.01±0.02	91.85±0.04	94.56±0.07 (2.71↑)
Refactory	**GGNN**	66.37±0.02	67.34±0.02	72.81±0.03 (5.47↑)
	GCN	64.07±0.01	64.32±0.03	67.73±0.03 (3.41↑)
	GAT	62.03±0.02	62.31±0.01	66.49±0.01 (4.38↑)
	CodeBERT	92.14±0.12	92.54±0.09	95.41±0.06 (2.87↑)
	GraphCodeBERT	92.02±0.05	92.15±0.03	95.23±0.11 (3.08↑)

MIXCODE can increase the model accuracy with an impressive improvement by up to 6.24%.

The lower part in Table III presents the robustness of different trained models on the transformed test data. First of all, from the test accuracy to the robustness, we observe that the results of all standard-trained models drop significantly, e.g., from 71% to 40% for JAVA250-BagofToken. This phenomenon confirms that only using original training data to train the model will result in a bad generalization property, i.e., the model can not handle the unseen data even if the data is functionally the same as the training data. However, the drop is reduced via data augmentation, especially by MIXCODE. MIXCODE achieves the best results in all cases, and the basic data augmentation usually performs similarly to the standard training. More specifically, the results show that it outperforms the basic data augmentation by up to 26.06%, and on average, 5.58%, which is remarkable. Particularly, in the problem classification task (JAVA250 and Python800), all the models have more than 20% accuracy improvement. And perhaps surprisingly, the robustness is already close to the original test accuracy, e.g., Python800-BagofToken, original accuracy 68.27% vs. robustness 63.65%. In the bug detection task (CodRep1 and Refactory), although the improvement is lower than in the problem classification (JAVA250 and Python800), it is still promising (from 2.87% to 5.47%).

Answer to RQ1: MIXCODE is effective in enhancing model performance. It outperforms the basic data augmentation by up to 6.24% and 26.06% original test accuracy and robustness improvement, respectively.

B. *RQ2: Impact of Mixup Strategies*

In the default setting of MIXCODE in step 2, a pair of randomly selected original code and transformed code is mixed (*Ori+Ref*). To investigate the usefulness of this mixing strategy, we compare (*Ori+Ref*) to other two types of Mixup strategies, original code mixing original code (*Ori+Ori*) and transformed code mixing transformed code (*Ref+Ref*).

Table IV shows the effectiveness comparison of these three strategies. First, compared to the results by producing standard training in Table III, we can see that all three strategies benefit the model performance. Meanwhile, in most cases (84 out of 108), the Mixup strategy outperforms the basic data augmentation. Next, we compare the effectiveness of different Mixup strategies. Considering the test accuracy, the results show that for the Java language (JAVA250, CodRep1), *Ori+Ref* consistently outperforms the other two combinations on all the tasks. *Ref+Ref*, the second-best, is slightly better than *Ori+Ori*. However, overall, the difference between these three strategies is small, e.g., 94.49% vs 94.52%, 93.18%. For the Python language (Python800, Refactory), we can find that there is no one can always be the best. But the *Ori+Ref* is still in the top-2 places in all cases. And even if *Ori+Ref* is in the second place, the gap between it and the best is slight (e.g., 84.12% vs 84.61%). Unlike Java tasks, the difference between these three combinations becomes bigger in Python Tasks, e.g., 82.46% vs 88.22% vs 86.58%. Thus, if we only consider the test accuracy, *Ori+Ref* is the recommended combination for MIXCODE. Considering the robustness, $Ori+Ref$ consistently and significantly outperforms the other two combinations with an average of 2.62% better robustness than the second best. These results recommend that using original data and transformed data is a better choice for MIXCODE to train robust models.

Then, we study how the hyperparameter α in Beta distribution for the determination of λ in Eq. (1) influences the performance of MIXCODE. The α can be set from 0 to ∞, but it is impractical to traverse all the possibilities. We follow the original Mixup setting where $\alpha = 0.2$ is the recommended setting and study α ranging from 0.05 to 0.5. Table V shows both the test accuracy and robustness of trained models. Note that when setting the α as 0.5, the SeqofToken-based models are always poor (with less than 1% accuracy). We conjecture that when α and λ become bigger, the mixed code is meaningless, and the model cannot learn anything from the data. A deeper analysis of this interesting phenomenon will be our future work. From the remaining results, we can see that MIXCODE can beat the standard training and basic data augmentation in most cases regardless of the setting of α. The results demonstrate that a smaller α can produce a better model. In 13 (out of 18) cases, and 15 (out of 18)

TABLE IV
COMPARISON OF DIFFERENT MIXUP STRATEGIES ON TEST ACCURACY AND ROBUSTNESS (AVERAGE± STANDARD DEVIATION, %). REF: TRANSFORMED CODE, ORI: ORIGINAL CODE. THE VALUE WITH A GRAY BACKGROUND INDICATES THE BEST RESULT. THE HIGHER, THE BETTER.

Dataset	DNN	Mixup Strategy		
		Ori+Ori	Ori+Ref	Ref+Ref
		Test Accuracy		
JAVA250	BagofToken	73.09±0.03	82.32±0.07	82.13±0.03
	SeqofToken	94.49±0.08	94.52±0.23	93.18±0.15
	CodeBERT	96.54±0.02	96.98±0.09	95.17±0.16
	GraphCodeBERT	96.76±0.04	97.16±0.12	95.56±0.19
CodRep1	GGNN	64.37±0.25	65.42±0.35	65.38±0.33
	GCN	63.42±0.26	64.18±0.44	63.89±0.27
	GAT	63.29±0.47	64.09±0.36	63.78±0.35
	CodeBERT	72.77±0.07	72.96±0.05	71.14±0.03
	GraphCodeBERT	73.21±0.11	73.34±0.13	71.78±0.16
Python800	BagofToken	68.27±0.08	67.88±0.07	65.67±0.09
	SeqofToken	84.68±0.04	85.00±0.20	81.22±0.45
	CodeBERT	96.35±0.04	96.79±0.06	95.11±0.18
	GraphCodeBERT	96.87±0.05	97.09±0.08	95.16±0.21
Refactory	GGNN	82.46±0.28	88.22±0.18	86.58±0.22
	GCN	81.71±0.24	84.12±0.17	84.61±0.25
	GAT	80.22±0.27	82.68±0.12	83.38±0.31
	CodeBERT	96.98±0.41	97.57±0.37	95.39±0.39
	GraphCodeBERT	97.78±0.27	98.16±0.21	96.03±0.19
		Robustness		
JAVA250	BagofToken	43.87±0.03	78.17±0.05	66.62±0.03
	SeqofToken	78.27±0.02	85.60±0.01	84.30±0.02
	CodeBERT	91.17±0.22	93.17±0.11	93.01±0.13
	GraphCodeBERT	92.01±0.19	94.07±0.16	93.56±0.17
CodRep1	GGNN	42.40±0.02	55.01±0.01	49.53±0.02
	GCN	42.08±0.02	54.03±0.03	49.42±0.03
	GAT	41.66±0.02	53.81±0.03	48.78±0.02
	CodeBERT	63.26±0.06	66.15±0.09	65.95±0.03
	GraphCodeBERT	63.17±0.08	65.91±0.13	64.76±0.16
Python800	BagofToken	40.46±0.01	63.65±0.02	62.35±0.03
	SeqofToken	75.15±0.02	82.94±0.02	78.54±0.01
	CodeBERT	90.08±0.05	92.16±0.08	92.08±0.03
	GraphCodeBERT	91.87±0.08	94.56±0.07	93.12±0.09
Refactory	GGNN	68.09±0.03	72.81±0.03	69.69±0.02
	GCN	65.68±0.02	67.73±0.03	66.36±0.01
	GAT	62.37±0.02	66.49±0.01	65.09±0.02
	CodeBERT	92.67±0.06	95.41±0.06	93.44±0.07
	GraphCodeBERT	92.45±0.09	95.23±0.11	93.17±0.13

cases, $\alpha = 0.05$ or $\alpha = 0.1$ can produce models with higher accuracy and better robustness.

Answer to RQ2: Compared to single-type (only original or transformed) code mixing, using both types (original and transformed) code is the best strategy for MIXCODE to train more accurate and robust models. A small α (e.g., 0.05 and 0.1) value is recommended for MIXCODE.

C. RQ3: Impact of Refactoring Methods

The refactoring method that determines the quality of transformed code is an important component in MIXCODE. In RQ1 and RQ2, we consider randomly selecting refactoring methods from all the possibilities to prepare the transformed data. However, it is still unclear how these refactor methods influence the performance of MIXCODE. Therefore, in this research question, we train the model by using each refactoring method separately under the best Mixup strategy *Ori+Ref* proved in RQ2 to rank the refactoring methods based on the performance of the trained model. Then, we evenly split the 18 refactoring methods based on the ranking into good (high ranking) and poor (low ranking), two sets. We only consider two types of combinations because it is hard to consider all the situations. Afterward, we perform MIXCODE again using these two sets, respectively. In this manner, we try to explore if there is a chance to further improve the MIXCODE by using a better refactoring method combination. Here, we choose two models for our study, BagofToken for the problem classification task and GGNN for the bug detection task. MIXCODE makes the best improvement in these two models, it is easier to amplify the difference.

Table VI presents the test accuracy of trained models using different individual program refactoring methods on the original test data. Surprisingly, MIXCODE using a single refactoring method can produce more accurate models than using all the refactoring methods, and the gap can be up to 4.92% (comparing *Arguments Adding* to *Baseline* in Java language). Then, the refactoring method with accuracy greater than 86.00%, 65.75%, 68.00%, and 89.00% from JAVA250, CodRep1, Python800, and Refactory is selected in the *Good*, the others are put in the *Poor*. We observe that considering only the test accuracy, compared to using *Poor*, using MIXCODE with refactoring methods from *Good* can train a more accurate model. On average, *Good* outperforms *Poor* with 1.98% test accuracy improvement. These results reveal that using a smartly selected single refactoring method is enough for MIXCODE if we only care about the test accuracy of the trained model. Additionally, combining better-refactoring methods (which have higher single-test accuracy) can build a more effective MIXCODE.

Moving to the robustness, table VII presents the results of trained models. First, compared with the standard training (in table III), MIXCODE with a single refactoring method can still produce more robust models. Only one case, *Local Variable Adding - Refactory*, has lower robustness than standard training. Then, we compare single-method and multiple-method combinations. Different from the pure test accuracy, the results show that MIXCODE with a single refactoring method can not outperform using multiple methods, and the difference gap is huge. For example, in JAVA250, the robustness of using a single method ranges from 42.42% to 68.34%, but the robustness of using *Poor* can be 78.19%, where around 10% robustness difference appeared. It is reasonable since if we force the model to learn one specific code refactoring, the generalization of the trained model could be quite low. Then, we compare *Good* and *Poor*. Surprisingly, in half of the cases (2 out of 4), *Poor* has higher robustness than *Good*. It is not the case that models trained by using better-refactoring methods (with higher original test accuracy) always have better robustness. This finding reveals a trade-off between original test accuracy and robustness when considering the refactoring methods for MIXCODE.

Answer to RQ3: Using a specific refactoring method (depending on the dataset and DNN), MIXCODE can produce models with high test accuracy but has to scarify the robustness. Utilizing multiple methods to enrich the diversity in data remains the best solution to adjust the trade-off between test accuracy and robustness.

VI. DISCUSSION

In this section, we discuss the limitation of our work, potential future research directions, and the threats to validity.

TABLE V
RESULTS OF MIXCODE USING ORI+REF STRATEGY FOR MIXUP WITH DIFFERENT α. THE VALUE WITH A GRAY BACKGROUND INDICATES THE BEST RESULT. THE HIGHER, THE BETTER.

Dataset	DNN	α =0.05	α =0.1	α =0.2	α =0.3	α =0.4	α =0.5	α =0.05	α =0.1	α =0.2	α =0.3	α =0.4	α =0.5
		Test Accuracy						Robustness					
JAVA	BagofToken	82.08±0.09	82.32±0.07	82.19±0.05	81.76±0.05	81.15±0.03	80.92±0.05	77.63±0.06	78.17±0.05	78.15±0.03	77.94±0.04	77.90±0.04	77.28±0.03
	SeqofToken	95.23±0.28	94.52±0.02	93.20±0.26	90.84±0.65	90.55±0.61	-	86.66±0.25	85.60±0.01	82.96±0.03	81.43±0.01	81.18±0.02	-
	CodeBERT	96.39±0.04	96.98±0.09	97.02±0.06	95.46±0.19	95.02±0.21	94.11±0.24	93.31±0.08	93.17±0.11	93.08±0.05	92.68±0.04	92.53±0.08	91.97±0.03
	GraphCodeBERT	97.03±0.14	97.16±0.12	97.14±0.16	96.03±0.21	96.32±0.24	95.81±0.29	94.28±0.11	94.07±0.16	93.87±0.12	93.13±0.17	94.01±0.11	92.83±0.09
CodReq1	GGNN	65.44±0.29	65.42±0.35	65.31±0.26	65.19±0.33	64.78±0.46	64.65±0.25	55.61±0.03	55.01±0.01	54.88±0.04	54.32±0.03	54.08±0.02	53.97±0.03
	GCN	64.12±0.31	64.18±0.44	64.03±0.34	63.89±0.28	63.76±0.38	63.56±0.24	54.81±0.01	54.03±0.03	53.97±0.04	53.42±0.05	53.08±0.02	52.78±0.03
	GAT	64.03±0.28	64.09±0.36	63.78±0.27	63.58±0.31	63.23±0.29	63.07±0.22	53.92±0.01	53.81±0.03	53.26±0.02	52.97±0.01	52.69±0.02	52.35±0.04
	CodeBERT	72.98±0.03	72.96±0.05	72.51±0.16	71.84±0.25	71.37±0.15	70.26±0.27	66.12±0.04	66.15±0.09	65.87±0.06	65.41±0.04	65.87±0.08	65.03±0.08
	GraphCodeBERT	73.11±0.14	73.34±0.13	72.67±0.19	72.19±0.22	71.62±0.25	70.34±0.29	66.23±0.11	65.91±0.13	65.36±0.18	64.78±0.21	64.85±0.17	64.21±0.21
Python800	BagofToken	67.14±0.09	67.88±0.07	68.22±0.06	68.61±0.05	68.53±0.08	68.72±0.06	63.31±0.05	63.65±0.02	64.58±0.01	64.74±0.03	64.82±0.02	64.73±0.01
	SeqofToken	84.47±0.26	85.00±0.20	84.31±0.05	83.50±0.05	82.89±0.34	-	83.53±0.16	82.94±0.02	82.17±0.01	81.24±0.02	80.81±0.03	-
	CodeBERT	96.82±0.11	96.79±0.06	96.04±0.11	95.22±0.18	94.87±0.31	94.08±0.23	92.11±0.07	92.16±0.08	92.09±0.09	91.87±0.07	91.44±0.09	91.01±0.03
	GraphCodeBERT	97.10±0.06	97.09±0.08	97.01±0.14	97.13±0.09	96.27±0.24	95.67±0.13	94.87±0.03	94.56±0.07	94.95±0.11	94.02±0.13	93.24±0.19	92.67±0.13
Refactory	GGNN	88.01±0.21	88.22±0.18	86.52±0.19	86.89±0.23	87.85±0.35	87.35±0.32	72.49±0.04	72.81±0.03	71.07±0.02	71.16±0.02	71.89±0.01	71.60±0.04
	GCN	84.10±0.16	84.12±0.17	84.08±0.26	84.06±0.33	83.63±0.25	82.98±0.29	67.91±0.01	67.73±0.03	67.38±0.02	67.32±0.04	66.93±0.03	66.04±0.01
	GAT	82.76±0.22	81.30±0.12	80.95±0.25	81.28±0.37	81.12±0.28	81.18±0.25	67.17±0.02	66.49±0.01	65.61±0.03	66.39±0.02	66.02±0.01	66.04±0.03
	CodeBERT	95.78±0.12	97.57±0.37	96.69±0.19	95.94±0.22	97.85±0.24	95.59±0.33	95.52±0.06	95.41±0.06	95.01±0.05	95.37±0.03	94.86±0.03	94.16±0.04
	GraphCodeBERT	98.03±0.24	98.16±0.21	98.47±0.25	98.23±0.27	97.92±0.21	97.44±0.23	95.31±0.13	95.23±0.11	95.17±0.17	94.21±0.27	95.53±0.31	94.07±0.29

TABLE VI
REFACTORING METHODS SELECTION (TEST ACCURACY). GOOD/POOR: MIXCODE USING THE BEST/WORST 9 REFACTORING METHODS, RESPECTIVELY. BASELINE: MIXCODE USING ALL 18 METHODS. THE BEST METHOD OF 18 FOR EACH DATASET IS HIGHLIGHTED WITH A GRAY BACKGROUND.

Refactoring Method	JAVA250	CodRep1	Python800	Refactory
	BagofToken	GGNN	BagofToken	GGNN
API Renaming	86.91±0.04	65.54±0.22	68.23±0.08	88.43±0.22
Arguments Adding	87.24±0.04	65.72±0.25	68.15±0.07	88.81±0.22
Argument Renaming	86.74±0.06	65.62±0.25	68.08±0.07	88.52±0.32
Dead For Adding	82.61±0.04	65.70±0.31	67.23±0.06	89.35±0.44
Dead If Adding	83.91±0.05	65.82±0.32	67.58±0.09	89.34±0.33
Dead If Else Adding	83.42±0.06	65.76±0.34	67.24±0.07	90.02±0.23
Dead Switch Adding	83.46±0.03	65.75±0.22	-	-
Dead While Adding	83.91±0.05	65.77±0.21	67.57±0.07	89.05±0.34
Duplication	85.61±0.06	65.44±0.33	67.32±0.06	88.36±0.21
Filed Enhancement	86.55±0.06	66.06±0.32	68.26±0.06	89.54±0.32
For Loop Enhancement	86.71±0.07	65.98±0.25	68.47±0.07	89.89±0.33
If Enhancement	86.35±0.03	66.35±0.23	68.26±0.05	91.04±0.33
Local Variable Adding	85.70±0.04	65.73±0.21	67.17±0.09	88.33±0.23
Local Variable Renaming	85.82±0.04	65.60±0.21	67.26±0.09	88.34±0.24
Method Name Renaming	87.02±0.05	65.04±0.43	68.59±0.06	88.32±0.22
Plus Zero	86.93±0.06	65.83±0.43	67.96±0.06	89.06±0.33
Print Adding	86.36±0.03	66.01±0.21	67.89±0.07	88.86±0.24
Return Optimal	85.85±0.06	65.53±0.34	67.28±0.08	88.33±0.22
Good (9)	86.34±0.06	65.72±0.24	68.03±0.06	89.97±0.17
Poor (9)	82.63±0.05	63.32±0.19	67.92±0.08	88.29±0.11
Baseline (18)	82.32±0.07	65.42±0.35	68.27±0.08	88.22±0.18

TABLE VII
REFACTORING METHODS SELECTION (ROBUSTNESS). GOOD/POOR: MIXCODE USING THE BEST/WORST 9 REFACTORING METHODS, RESPECTIVELY. BASELINE: MIXCODE USING ALL 18 METHODS. THE BEST METHOD OF 18 FOR EACH DATASET IS HIGHLIGHTED WITH A GRAY BACKGROUND.

Refactoring Method	JAVA250	CodRep1	Python800	Refactory
	BagofToken	GGNN	BagofToken	GGNN
API Renaming	43.61±0.02	55.06±0.02	40.11±0.02	66.65±0.02
Arguments Adding	42.72±0.03	55.09±0.01	39.05±0.01	66.68±0.02
Argument Renaming	43.27±0.04	55.04±0.01	40.08 ±0.01	66.58±0.01
Dead For Adding	45.45±0.02	55.15±0.02	45.43±0.04	68.10±0.01
Dead If Adding	68.34±0.03	55.16±0.03	56.74±0.01	68.20±0.03
Dead If Else Adding	68.27±0.01	55.19±0.02	56.34±0.02	68.34±0.02
Dead Switch Adding	48.71±0.03	55.11±0.01	-	-
Dead While Adding	65.33±0.02	55.17±0.03	55.44±0.02	68.09±0.02
Duplication	43.29±0.03	55.03±0.03	40.88±0.02	66.67±0.01
Filed Enhancement	44.11±0.01	55.10±0.04	42.14 ±0.02	66.59±0.02
For Loop Enhancement	42.42±0.03	55.09±0.02	41.29±0.02	66.64±0.02
If Enhancement	42.46±0.04	55.05±0.02	41.03±0.02	66.39±0.03
Local Variable Adding	43.34±0.02	55.08±0.02	40.78±0.03	66.32±0.02
Local Variable Renaming	44.44±0.04	55.09±0.03	40.67±0.02	66.34±0.01
Method Name Renaming	44.31±0.01	55.06±0.02	41.13±0.02	66.64±0.02
Plus Zero	43.16±0.01	55.06±0.02	40.65±0.02	66.43±0.02
Print Adding	44.27±0.02	55.03±0.04	41.08±0.03	66.67±0.02
Return Optimal	42.63±0.03	55.08±0.02	39.87±0.02	66.47±0.01
Good (9)	43.50±0.03	55.21±0.03	56.47±0.02	72.91±0.02
Poor (9)	78.19±0.02	52.09±0.02	63.55±0.01	71.47±0.01
Baseline (18)	78.17±0.05	55.01±0.01	63.65±0.02	72.81±0.03

A. Limitation & Future Directions

Limitation. In Section V-B, we observed that the setting of hyperparameter α could affect the performance of MIXCODE. And in the worst case, e.g., $\alpha = 0.5$, some models can learn nothing from the mixed data. The potential reason is still unclear, raising concerns about using MIXCODE. However, it is possible to bypass this limitation by using MIXCODE with a smaller α.

Future directions. Existing works for source code analysis mainly focus on proposing new code representation and code embedding approaches [8], [35] or studying how to utilize large pre-trained language models for downstream code tasks [55]. Only limited works consider the importance of the quality of training data [56] or the importance of training strategies. There are still many opportunities in source code analysis for better program coding. We highlight two potential research directions for future exploration.

1) In terms of MIXCODE, we investigate the impact of different refactoring methods and combinations on both the test accuracy and robustness of FNN, CNN, GGNN, GCN, GAT, CodeBERT, and GraphCodeBERT. Despite that MIXCODE brings better performance, there is still room to improve. The straightforward research direction is to design an adaptive method to find the best combination of refactoring methods.

2) In terms of Mixup, a series of works propose different variants of the original Mixup. Therefore, in addition to using raw code vectors to do code mixing, other options, such as the raw source code and the embedding vectors, can also be used as the input of the Mixup approach.

B. Threats to Validity

The internal threat to validity comes from the implementation of standard training, basic data augmentation, and MIXCODE. The training for the classification tasks (JAVA250, Python800) is taken from the project CodeNet [5], and the defect detection tasks (CodRep1, Refactory) adopt from [48], [49], [57]. The implementation of Mixup comes from its original release [20]. The 18 refactoring methods for the Java language are from [14], [43], and we adapt the implementation to the Python language.

The external threats to validity lie in the selected source code tasks, datasets, DNNs, and refactoring methods. We consider two different tasks (problem classification and bug detection) in the study and include two datasets for each task. Particularly, we include two popular programming languages (Java and Python) for software developers. Remarkably, we utilize seven types of deep neural networks, including the most famous pre-trained programming language models. For the refactoring methods, we cover the most common ones from the literature.

The construct threats to validity mainly come from the parameters of MixCode, randomness, and evaluation measures. MixCode only contains the parameter λ that controls the weight of mixing two input instances. We follow the recommendation of the original Mixup algorithm and investigate the impact of this parameter. We observe that regardless of the parameter setting, MixCode still outperforms the standard training and basic data augmentation. To reduce the impact of randomness, we repeat each experiment five times and report the average and standard deviation results. Finally, for evaluation measures, we consider both the accuracy of the original test data and the robustness of transformed test data. The latter one is specific for evaluating the generalization ability of DNNs.

VII. Related work

We consider related works from three aspects, source code learning enhancement, data augmentation for source code analysis, and Mixup for image and text classification.

A. Source Code Learning Enhancement

The goal of our work is to improve the performance of source code-related models. Existing works targeting the same goal try to challenge this problem in different ways.

Self-supervised learning. Although the code model has significantly succeeded in software engineering applications, there are still some limitations. For example, some code models are built for solving a particular problem, thus, the related code representation is difficult to extend to other problems. To enhance the flexibility of code models, Bui et al. [58] proposed to utilize a self-supervised learning mechanism for code representation preparation. Specifically, it utilizes the identified sub-trees prediction as the self-supervised task to train the code representation. As a result, the code representation can be used in different downstream tasks.

Pre-trained models fine-tuning. Reusing pre-trained models to solve code analysis problems is another straightforward method. Multiple models have been proposed. Kanade et al. [59] introduced CuBERT, the same model architecture as BERT [60], trained on Python source code. Buratti et al. [61] used a transformer-based model that was trained on C language to build the pre-trained model C-BERT. Here, CuBERT and C-BERT were both trained by using a single programming language. More recently, some multi-programming language pre-trained models were proposed. Lu et al. [53] provided CodeGPT, which was trained on Python and Java corpora. Feng et al. [7] presented CodeBERT, a pre-trained model that learns information from six programming languages and natural language text. To further capture the semantic structure of code, Guo et al. [35] proposed GraphCodeBERT, which uses data flow information of code in the pre-training stage. Different from these works, MixCode mainly focuses on improving the model training process from the perspective of data augmentation. As a result, MixCode can be generalized to any code classification task regardless of pre-trained models.

B. Data Augmentation for Source Code Analysis

Data augmentation has achieved tremendous success in the CV and NLP fields [10], [11]. Recently, due to the similar processing workflow of NLP data and source code, researchers devoted considerable effort to applying this technique to improve the performance of the code model. The widely studied data augmentation method, adversarial training [62], has been studied in code learning. Simply, adversarial training generates a set of adversarial examples and combines them with the original training data to increase the data volume. For instance, Zhang et al. [63] generated adversarial examples based on the Metropolis-Hastings modifier (MHM) algorithm [64]. Mi et al. [65] generated the synthetic data from Auxiliary Classifier generative adversarial networks (GANs) to increase the data size. Unlike the above works, we generate new data by refactoring the programs and then apply Mixup to enrich the volume and diversity of the training dataset.

C. Mixup for CV and NLP

Compared to the above-mentioned data augmentation methods that increase the data volume by combining adversarial examples and the original training data, Mixup [20] linearly mixes existing training data to increase the diversity of learned information by the model. Recently, multiple variants of Mixup have been proposed [41], [66]–[74]. Yun et al. [75] applied Dropout [76] into Mixup, and proposed a mixing strategy based on the patch of the image. Besides, Liu et al. [77] introduced the Automatic Mixup (AutoMix) strategy to balance the mixing policies and optimization complexity. Although the original Mixup is proposed for image data, researchers have extended the Mixup to support other types of data. For text data, Yoon et al. [78] synthesized the new text data from two raw input data by replacing the hidden vectors based on span-based mixing. Wang et al. [79] provided a two-stage Mixup framework for graph data. One is mixing the feature of node neighbors, and another is mixing the feature of the entire graph. Like Mixup for text and graph data classification, we augment the input code data in the vector space for source code classification. However, the difference is that we consider the code with sequence and graph representation, and we mix the data with their transformed version.

VIII. Conclusion

This paper presented MixCode, the first data augmentation framework for source code analysis, to enhance model performance without collecting or labeling new code. Specifically, MixCode supports 18 types of refactoring methods (extensible with new ones) that generate transformed code. To evaluate the effectiveness of MixCode, we conducted extensive experiments on two important code tasks (problem classification and bug detection) and seven DNN architectures. Experimental results demonstrated MixCode outperforms the basic data augmentation baseline by up to 6.24% accuracy and 26.06% robustness improvement. The configuration study proved that linearly mixing the original and transformed code achieves the best performance of MixCode.

IX. Acknowledgement

This research is supported in part by JSPS KAKENHI Grant No. JP19H04086, Japan. Qiang Hu is also supported by the Luxembourg National Research Funds (FNR) through CORE project C18/IS/12669767/STELLAR/LeTraon.

References

[1] M. Wang and W. Deng, "Deep face recognition: A survey," *Neurocomputing*, vol. 429, pp. 215–244, 2021.

[2] D. Dong, H. Wu, W. He, D. Yu, and H. Wang, "Multi-task learning for multiple language translation," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 1723–1732.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[4] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[5] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[6] Y. Shi, T. Mao, T. Barnes, M. Chi, and T. W. Price, "More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code." in *In Proceedings of the 14th International Conference on Educational Data Mining (EDM) 2021*, 2021.

[7] 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.

[8] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, "Graphcode2vec: Generic code embedding via lexical and program dependence analyses," *arXiv preprint arXiv:2112.01218*, 2021.

[9] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[10] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[11] S. Y. Feng, V. Gangal, J. Wei, S. Chandar, S. Vosoughi, T. Mitamura, and E. Hovy, "A survey of data augmentation approaches for nlp," *arXiv preprint arXiv:2105.03075*, 2021.

[12] T. Zhao, Y. Liu, L. Neves, O. Woodford, M. Jiang, and N. Shah, "Data augmentation for graph neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, 2021, pp. 11 015–11 023.

[13] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[14] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 36–46.

[15] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[16] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[17] 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.

[18] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *Journal of Systems and Software*, vol. 190, p. 111304, 2022.

[19] P. Bielik and M. Vechev, "Adversarial robustness for code," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 896–907. [Online]. Available: https://proceedings.mlr.press/v119/bielik20a.html

[20] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.

[21] H. Xu and S. Mannor, "Robustness and generalization," *Machine learning*, vol. 86, no. 3, pp. 391–423, 2012.

[22] K. Kawaguchi, L. P. Kaelbling, and Y. Bengio, "Generalization in deep learning," *arXiv preprint arXiv:1710.05468*, 2017.

[23] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[24] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[25] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 60–70.

[26] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 39–40.

[27] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[28] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 799–10 808.

[29] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[30] Z. Chen, V. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, "Plur: A unifying, graph-based view of program learning, understanding, and repair," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[31] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[32] L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. Byrd, M. Might, R. Urtasun, and R. Zemel, "Neural guided constraint logic programming for program synthesis," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[33] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[34] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[35] 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.

[36] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *arXiv preprint arXiv:1711.00740*, 2017.

[37] P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani, W. Hu, M. Yasunaga, R. L. Phillips, I. Gao *et al.*, “Wilds: A benchmark of in-the-wild distribution shifts,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 5637–5664.

[38] Y. Zhu, T. Ko, and B. Mak, “Mixup learning strategies for text-independent speaker verification.” in *Interspeech*, 2019, pp. 4345–4349.

[39] A. Kaur and M. Kaur, “Analysis of code refactoring impact on software quality,” in *MATEC Web of Conferences*, vol. 57. EDP Sciences, 2016, p. 02012.

[40] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[41] H. Guo, Y. Mao, and R. Zhang, “Augmenting data with mixup for sentence classification: An empirical study,” *arXiv preprint arXiv:1905.08941*, 2019.

[42] J. B. McDonald and Y. J. Xu, “A generalization of the beta distribution with applications,” *Journal of Econometrics*, vol. 66, no. 1-2, pp. 133–152, 1995.

[43] M. Wei, Y. Huang, J. Yang, J. Wang, and S. Wang, “Cocofuzzing: Testing neural code models with coverage-guided fuzzing,” *arXiv preprint arXiv:2106.09242*, 2021.

[44] T. H. M. Le, H. Chen, and M. A. Babar, “Deep learning for source code modeling and generation: models, applications, and challenges,” *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3383458

[45] D. Hendrycks, N. Mu, E. D. Cubuk, B. Zoph, J. Gilmer, and B. Lakshminarayanan, “Augmix: A simple data processing method to improve robustness and uncertainty,” *arXiv preprint arXiv:1912.02781*, 2019.

[46] D. Hendrycks, A. Zou, M. Mazeika, L. Tang, B. Li, D. Song, and J. Steinhardt, “Pixmix: Dreamlike pictures comprehensively improve safety measures,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 16 783–16 792.

[47] L. Zhang, Z. Deng, K. Kawaguchi, A. Ghorbani, and J. Zou, “How does mixup help with robustness and generalization?” *arXiv preprint arXiv:2010.04819*, 2020.

[48] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 388–398.

[49] H. Zhong and Z. Su, “An empirical study on real bug fixes,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[50] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” *arXiv preprint arXiv:1511.05493*, 2015.

[51] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[52] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” *stat*, vol. 1050, p. 20, 2017.

[53] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.

[54] X. Zhou, D. Han, and D. Lo, “Assessing generalizability of codebert,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[55] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[56] Z. Sun, L. Li, Y. Liu, and X. Du, “On the importance of building high-quality training datasets for neural code search,” *arXiv preprint arXiv:2202.06649*, 2022.

[57] Z. Chen and M. Monperrus, “The codrep machine learning on source code competition,” *arXiv preprint arXiv:1807.03200*, 2018.

[58] N. D. Bui, Y. Yu, and L. Jiang, “Infercode: Self-supervised learning of code representations by predicting subtrees,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[59] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[60] 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.

[61] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, “Exploring software naturalness through neural language models,” *arXiv preprint arXiv:2006.12641*, 2020.

[62] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” *arXiv preprint arXiv:1412.6572*, 2014.

[63] X. Zhang, Y. Zhou, T. Han, and T. Chen, “Training deep code comment generation models via data augmentation,” in *12th Asia-Pacific Symposium on Internetware*, 2020, pp. 185–188.

[64] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, “Generating adversarial examples for holding robustness of source code processing models,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1169–1176.

[65] Q. Mi, Y. Xiao, Z. Cai, and X. Jia, “The effectiveness of data augmentation in code readability classification,” *Information and Software Technology*, vol. 129, p. 106378, 2021.

[66] L. Sun, C. Xia, W. Yin, T. Liang, P. S. Yu, and L. He, “Mixup-transformer: dynamic data augmentation for nlp tasks,” *arXiv preprint arXiv:2010.02394*, 2020.

[67] J. Chen, Z. Yang, and D. Yang, “Mixtext: Linguistically-informed interpolation of hidden space for semi-supervised text classification,” *arXiv preprint arXiv:2004.12239*, 2020.

[68] R. Zhang, Y. Yu, and C. Zhang, “Seqmix: Augmenting active sequence labeling via sequence mixup,” *arXiv preprint arXiv:2010.02322*, 2020.

[69] D. Walawalkar, Z. Shen, Z. Liu, and M. Savvides, “Attentive cutmix: An enhanced data augmentation approach for deep learning based image classification,” *arXiv preprint arXiv:2003.13048*, 2020.

[70] A. Uddin, M. Monira, W. Shin, T. Chung, S.-H. Bae *et al.*, “Saliencymix: A saliency guided data augmentation strategy for better regularization,” *arXiv preprint arXiv:2006.01791*, 2020.

[71] J. Qin, J. Fang, Q. Zhang, W. Liu, X. Wang, and X. Wang, “Resizemix: Mixing data with preserved object information and true labels,” *arXiv preprint arXiv:2012.11101*, 2020.

[72] J.-H. Kim, W. Choo, and H. O. Song, “Puzzle mix: Exploiting saliency and local statistics for optimal mixup,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 5275–5285.

[73] J.-H. Kim, W. Choo, H. Jeong, and H. O. Song, “Co-mixup: Saliency guided joint mixup with supermodular diversity,” *arXiv preprint arXiv:2102.03065*, 2021.

[74] V. Verma, A. Lamb, C. Beckham, A. Najafi, I. Mitliagkas, D. Lopez-Paz, and Y. Bengio, “Manifold mixup: Better representations by interpolating hidden states,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 6438–6447.

[75] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, “Cutmix: Regularization strategy to train strong classifiers with localizable features,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6023–6032.

[76] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[77] Z. Liu, S. Li, D. Wu, Z. Chen, L. Wu, J. Guo, and S. Z. Li, “Unveiling the power of mixup for stronger classifiers,” *arXiv preprint arXiv:2103.13027*, 2021.

[78] S. Yoon, G. Kim, and K. Park, “Ssmix: Saliency-based span mixup for text classification,” *arXiv preprint arXiv:2106.08062*, 2021.

[79] Y. Wang, W. Wang, Y. Liang, Y. Cai, and B. Hooi, “Mixup for node and graph classification,” in *Proceedings of the Web Conference 2021*, 2021, pp. 3663–3674.