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, {zheng, zhaof}@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
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.\footnote{https://github.com/zemingd/Mixup4Code}
- 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
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
Ensure: M: trained model

Standard training (without augmentation)

1: for run ∈ {0, . . . , #epochs} do
2: M.Fit (X, Y)
3: return M

Basic augmentation

4: for run ∈ {0, . . . , #epochs} do
5: Xref ← ∅
6: for x ∈ X do
7: r ← RandomSelection (R)
8: Xref ← Xref ∪ r (x)
9: M.Fit (Xref, 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 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, y) and (x', y'), where x represents the input feature and its corresponding output label y is donated with one-hot encoding, Mixup produces new data pairs (x_mix, y_mix):

\[ \begin{align}
    x_{mix}^i &= \lambda x^i + (1 - \lambda) x'^i \\
    y_{mix}^i &= \lambda y^i + (1 - \lambda) y'^i
\end{align} \]

where \( \lambda \) is a mixing policy for the input sample pair, which is sampled from a \( Beta \) distribution with a shape parameter \( \alpha \) (\( \lambda \sim Beta(\alpha, \alpha) \)). 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.

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

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.](image)
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 \( \text{run} \leftarrow \{0, \ldots, \#\text{epochs}\} \) do
2: \( X_{\text{ref}}, X_{\text{mix}}, Y_{\text{mix}} \leftarrow \phi, \phi, \phi \) \quad \triangleright \text{initialization}
3: \( r \leftarrow \text{RandomSelection} (R) \)
4: \( X_{\text{ref}} \leftarrow X_{\text{ref}} \cup r (x) \) \quad \triangleright \text{code refactoring}
5: \( X_s, Y_s \leftarrow \text{Shuffle} (X, Y) \) \quad \triangleright \text{Shuffle the training set}
6: \( (x_s, y_s, x_{\text{ref}}, y) \in (X_s, Y_s, X_{\text{ref}}, Y) \) do
7: \( v_x \leftarrow T (x_s) \) \quad \triangleright \text{code representation}
8: \( v_{\text{ref}} \leftarrow T (x_{\text{ref}}) \) \quad \triangleright \text{code representation}
9: \( \lambda \leftarrow \text{Beta} (\alpha) \) \quad \triangleright \text{hyperparameter}
10: \( x_{\text{mix}} \leftarrow \lambda v_x + (1 - \lambda) v_{\text{ref}} \) \quad \triangleright \text{data generation}
11: \( y_{\text{mix}} \leftarrow \lambda y_s + (1 - \lambda) y \) \quad \triangleright \text{label generation}
12: \( Y_{\text{mix}} \leftarrow x_{\text{mix}} \cup x_{\text{mix}} \)
13: \( Y_{\text{mix}} \leftarrow y_{\text{mix}} \cup y_{\text{mix}} \)
14: \( M.\text{Fit} (X_{\text{mix}}, Y_{\text{mix}}) \) \quad \triangleright \text{training with augmented data}
15: 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 \( v_{\text{ref}} \), 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 \((x_{\text{mix}}, y_{\text{mix}})\) in a similar way as the original Mixup does, and add them into the training set \((X_{\text{mix}}, Y_{\text{mix}})\).

3) Finally, the mixed data set \((X_{\text{mix}}, Y_{\text{mix}})\) 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]. Simi-

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

```
for i in range(1, 10):
    for j in range(1, i + 1):
        print(f"{i}x{j} = {i*j}"),
    print()
```

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

- MIXCODE supports both Integer-type and Float-type input data by simply setting the input type of embedding layers as float (e.g., \( x = \text{torch.tensor} \text{(dataset, dtype=torch.float)} \)).

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 \( \alpha \). 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 \( \alpha \) controls the percentage (\( \lambda \)) of code content used from two parts for Mixup. As mentioned in Section II, \( \lambda \) is a value with the [0, 1] range and is sampled from a Beta distribution [42] parameterized by \( \alpha \) 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 \( \alpha \) 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
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 $\lambda$ in Eq. (1) as 0.2 and perform the input mixing.

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.

| 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 argument by a synonym of its name.                                | `def func(number) $\rightarrow$ def func(size)`            |
| 4   | Dead for adding          | Add an unreachable loop at a randomly selected location.                    | `for i in range (0) ; print (0)`                          |
| 5   | Dead if adding           | Add an unreachable if statement at a randomly selected location in the code. | `if (1 == 0) : print (0)`                                 |
| 6   | Dead if else adding      | Add an unreachable if-else statement at a randomly selected location in the code. | `print (0) if (1 == 0) else print (1)`                     |
| 7   | Dead switch adding       | Add an unreachable switch statement at a randomly selected location.        | `if a = 0, switch (a) case 1; break; default: System.out.println(“pass”);` |
| 8   | Dead while adding        | Add an unreachable while loop at a randomly selected location.              | `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 (“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.                                              | `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.                          | `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`      |

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

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

| 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, 50000, 25000                                                                                                       |
| Python800 Python Problem classification| 153600, 48000, 24000                                                                                                         |
| CodRep1 Java Bug detection            | 6944, 772, 7716                                                                                                               |
| Refactory Python Bug detection        | 3380, 422, 4225                                                                                                               |
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, we 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 convolutional 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 (averages ± 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 |
|--------------|-----------|----------|-------|---------|
| JAVA250      | BagoToken | 94.62±0.3 | 94.63±0.08 | 62.35±0.05 | 96.37±0.05 |
|              | SeqoToken | 86.57±0.06 | 93.92±0.15 | 94.52±0.23 | 60.60±0.07 |
|              | CodeBERT  | 96.37±0.02 | 96.46±0.07 | 96.89±0.09 | 95.02±0.07 |
|              | GraphCodeBERT | 96.48±0.03 | 96.51±0.06 | 97.16±0.12 | 96.85±0.06 |
| CodRep1      | GCN       | 62.09±0.32 | 65.39±0.19 | 65.42±0.35 | 2.52±0.02 |
|              | GAT       | 61.27±0.26 | 62.07±0.33 | 64.09±0.36 | 2.02±0.02 |
|              | CodeBERT  | 69.12±0.03 | 71.23±0.02 | 72.96±0.05 | 1.74±0.02 |
|              | GraphCodeBERT | 70.05±0.11 | 71.35±0.09 | 73.54±0.13 | 1.99±0.02 |
| Python800    | BagoToken | 67.51±0.05 | 67.46±0.08 | 68.27±0.08 | 0.81±0.1 |
|              | SeqoToken | 82.85±0.32 | 83.26±0.41 | 83.80±0.20 | 1.74±0.1 |
|              | CodeBERT  | 96.05±0.02 | 96.16±0.01 | 96.79±0.06 | 0.63±0.02 |
|              | GraphCodeBERT | 96.27±0.05 | 96.29±0.03 | 97.09±0.08 | 0.80±0.02 |
| Refactory    | GCN       | 80.12±0.25 | 80.23±0.37 | 84.16±0.25 | 1.93±0.01 |
|              | GAT       | 79.76±0.19 | 80.05±0.28 | 83.80±0.31 | 3.37±0.01 |
|              | CodeBERT  | 95.88±0.42 | 96.09±0.35 | 97.57±0.04 | 2.74±0.02 |
|              | GraphCodeBERT | 96.81±0.17 | 96.92±0.11 | 98.16±0.21 | 1.24±0.01 |

**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 α = 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 SeqoToken-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 |
|----------|--------------|----------------|
|          | Test Accuracy |                |
|          | Ori+Ori      | Ori+Ref        | Ref+Ref |
| JAV250   | BagofToken   | 73.09±0.03     | 82.32±0.07 | 82.13±0.03 |
|          | Seq2Token    | 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 |
|          | GNN          | 64.37±0.25     | 65.42±0.35 | 65.38±0.33 |
|          | GCN          | 63.42±0.26     | 64.16±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    | 68.27±0.05     | 67.81±0.07 | 65.67±0.09 |
|          | Seq2Token    | 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     | 97.00±0.08 | 95.16±0.21 |
| Refactory| GNN          | 82.46±0.28     | 82.22±0.18 | 86.58±0.22 |
|          | GCN          | 81.71±0.24     | 81.12±0.19 | 84.61±0.25 |
|          | GAT          | 80.22±0.27     | 82.66±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   |                |            |
| JAV250   | BagofToken   | 43.87±0.01     | 78.17±0.05 | 66.62±0.03 |
|          | Seq2Token    | 78.27±0.02     | 85.00±0.01 | 84.30±0.02 |
|          | CodeBERT     | 91.17±0.02     | 93.16±0.11 | 93.01±0.13 |
|          | GraphCodeBERT| 92.01±0.19     | 94.07±0.16 | 93.56±0.17 |
|          | GNN          | 42.49±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     | 63.68±0.02 | 62.35±0.01 |
|          | Seq2Token    | 75.15±0.02     | 82.94±0.02 | 78.54±0.01 |
|          | CodeBERT     | 90.06±0.05     | 92.16±0.08 | 92.08±0.03 |
|          | GraphCodeBERT| 91.87±0.08     | 94.56±0.07 | 93.12±0.09 |
|          | GNN          | 68.09±0.03     | 72.81±0.03 | 69.09±0.02 |
|          | GCN          | 65.68±0.02     | 67.33±0.03 | 66.36±0.01 |
|          | GAT          | 62.37±0.02     | 66.49±0.01 | 65.06±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 $\alpha$ (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 refactoring 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 smarter 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 cannot 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 sacrifice 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 |
|---------|-------------|----------|----------|----------|----------|----------|----------|
| JAVA    | BugToken    | 80.26±0.00 | 82.38±0.00 | 83.73±0.02 | 84.19±0.00 | 84.13±0.02 | 83.72±0.00 |
| SeqToken | 95.23±0.01 | 94.52±0.02 | 93.20±0.26 | 90.84±0.05 | 90.51±0.01 | 90.14±0.04 | 90.33±0.01 |
| CodeBERT | 96.30±0.01 | 95.46±0.01 | 94.06±0.05 | 92.86±0.04 | 92.53±0.08 | 91.97±0.03 |
| GraphCodeBERT | 97.03±0.14 | 96.16±0.25 | 95.13±0.20 | 93.62±0.24 | 93.01±0.29 | 91.98±0.24 |
| GCNN     | 65.49±0.40 | 64.42±0.36 | 63.15±0.26 | 63.19±0.03 | 64.78±0.46 | 64.85±0.25 |
| CodRepl  | 64.03±0.28 | 64.09±0.36 | 63.78±0.27 | 63.58±0.31 | 63.23±0.29 | 63.07±0.22 |
| GAT      | 95.90±0.01 | 95.86±0.02 | 95.21±0.16 | 94.84±0.25 | 93.71±0.15 | 92.76±0.27 |
| CodeBERT | 73.11±0.14 | 73.34±0.13 | 72.67±0.19 | 71.82±0.22 | 71.62±0.25 | 70.34±0.29 |
| GraphCodeBERT | 71.33±0.11 | 70.05±0.13 | 69.63±0.18 | 68.74±0.21 | 68.45±0.17 | 64.21±0.21 |
| Python800 | 84.47±0.26 | 85.05±0.20 | 84.31±0.05 | 83.50±0.06 | 82.89±0.43 | 82.14±0.34 |
| CodeBERT | 94.74±0.01 | 93.82±0.01 | 93.22±0.11 | 92.58±0.31 | 91.80±0.23 | 90.48±0.21 |
| GraphCodeBERT | 97.10±0.06 | 97.09±0.08 | 97.01±0.14 | 96.92±0.24 | 96.57±0.13 | 96.00±0.11 |
| GCNN     | 89.91±0.21 | 89.22±0.18 | 88.42±0.19 | 87.89±0.23 | 87.38±0.24 | 86.70±0.18 |
| Refactory | 82.96±0.22 | 82.97±0.22 | 82.61±0.23 | 82.09±0.24 | 82.00±0.25 | 81.59±0.26 |
| GAT      | 89.79±0.22 | 89.62±0.30 | 89.50±0.23 | 88.21±0.27 | 87.32±0.26 | 86.30±0.20 |
| CodeBERT | 95.76±0.12 | 95.77±0.37 | 96.69±0.19 | 95.84±0.22 | 95.50±0.33 | 94.57±0.23 |
| GraphCodeBERT | 99.03±0.24 | 98.16±0.21 | 97.45±0.25 | 96.83±0.27 | 96.41±0.23 | 95.46±0.21 |

**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., α = 0.5, some models can learn nothing from the mixed data. The potential reason is still unclear, raising confusing 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 focused on the importance of training data [56] or the importance of training strategies. There are still many opportunities in code analysis for better program coding. We highlight two potential 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, GCNN, GCN, GAT, CodeBERT, and GraphCodeBERT. Despite that MIXCODE brings better performance, there is still room to improve. The straightforward refactoring strategy 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 (CodRepl, Refactory) adopt from [48], [49], [57]. The implementation of Mixup comes from its original release [20]. The refering 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 \( \lambda \) 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, Bu 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/1266976/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. Sorensen, 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. Gangoth, J. Wei, S. Chandar, S. Bousoughi, 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, “Bringing 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. Wu, 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, “Dlix: 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, “Sequence: 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. Manzogol, D. Tarlov, 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. Hu, 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. 200–2010.
[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.
