Boosting Source Code Learning with Text-Oriented Data Augmentation: An Empirical Study

Zeming Dong[1], Qiang Hu[2], Yuejun Guo[3], Zhenya Zhang[1] and Jianjun Zhao[1]
[1]Kyushu University, Japan
[2]University of Luxembourg, Luxembourg
[3]Luxembourg Institute of Science and Technology, Luxembourg
dong.zeming.011@s.kyushu-u.ac.jp, qiang.hu@uni.lu, yuejun.guo@list.lu, {zhang, zhao}@ait.kyushu-u.ac.jp

Abstract—Recent studies have shown surprising results of *source code learning*, which applies *deep neural networks (DNNs)* to various software engineering tasks. Like other DNN-based domains, source code learning also requires massive high-quality training data to achieve the success of these applications. In practice, data augmentation is a technique that produces additional training data to boost the model training and has been widely adopted in other domains (e.g. *computer vision*). However, the existing practice of data augmentation in source code learning is limited to simple syntax-preserved methods, such as code refactoring. In this paper, based on the insight that source code can be represented sequentially as text data, we take an early step to investigate whether data augmentation methods originally for texts are effective for source code learning. To that end, we focus on code classification tasks and conduct a comprehensive empirical study on four critical code problems and four DNN architectures to assess the effectiveness of 8 data augmentation methods. Our results identify the data augmentation methods that can produce more accurate models for source code learning and show that the data augmentation methods are still useful even if they slightly break the syntax of source code.

Keywords–Data Augmentation, Source Code Analysis, Program Transformation

1. Introduction

In recent years, applying machine learning (ML) in the domain of big code (*ML4Code*) [1] has gained significant attention. ML4Code leverages the power of ML, especially *deep learning (DL)*, to extract patterns from large code corpora. The remarkable performance of ML4Code in various downstream code tasks, such as *clone detection* [2], *bug detection* [3], and *problem classification* [4], demonstrate its immense potential in facilitating developers in daily activity.

Well-designed model architectures and high-quality training data are essential factors in producing programming language (PL) models with outstanding performance. In practice, the model architecture has been extensively studied by researchers and many models can achieve state-of-the-art performance, e.g., GraphCodeBERT [5]) and CodeBERT [6]. However, preparing sufficient labeled data for model training remains an open challenge. The challenge comes from the expensive human efforts required for collecting, cleaning and annotating data. For example, annotating only four libraries of code can take 600 man-hours [7].

To address the issue of data scarcity, integrating data augmentation into training can be a promising solution. Essentially, data augmentation generates new training data by modifying existing labeled data under the premise that these new data preserve the original semantics. For example, when training an image classification model, instead of using the original training data only, a common practice is to utilize image transformation techniques [8] to produce more diverse images. While data augmentation has been well-studied and proven to be effective in other domains, such as *computer vision (CV)* and *natural language processing (NLP)*, its application in source code learning has received limited attention and its potential has not been fully exploited. An early trial was performed in [9], where a number of *code refactoring* methods were introduced for generating new code data. However, as reported in [9], the performance of those methods is limited. In this work, to bridge this gap, we investigate new data augmentation methods to improve the performance of model training in source code learning.

Contributions: In source code learning, raw code is often converted into machine-readable data format by representing it as a sequence of text tokens due to its analogy to texts [1], where each token is represented by an integer and then fed into the code model. Inspired by this token-based representation, we empirically study the problem of whether existing data augmentation approaches in NLP (that handles text data) are effective in improving the training quality in source code learning. Concretely, we first survey and categorize existing data augmentation methods in the literature, and we find seven data augmentation methods that are applicable to the source code. Then, we adapt these methods to train code models and investigate their effectiveness in improving the accuracy of those models. Overall, our study considers two mainstream programming languages (Java and Python), four crucial downstream classification tasks (problem classification, bug detection, authorship attribution, and clone detection), and four DNN model architectures including two pre-trained PL models (CodeBERT and GraphCodeBERT). In total, 1,080 models have been trained and studied in our work.

We design our large-scale study in order to answer the following research questions:

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00017

RQ1: Can existing data augmentation methods produce accurate code models? The results show that data augmentation methods that linearly mix feature vectors in code embedding, e.g., *SenMixup*, can enhance the accuracy by up to 8.74%, compared to the training without using data augmentation. Remarkably, the methods adapted from NLP are more effective than the code-specific data augmentation technique, namely, code refactoring.

RQ2: How does data volume affect the effectiveness of data augmentation methods? The results demonstrate that when training data is scarce, incorporating data augmentation can help to improve the accuracy. For example, using *SenMixup* can improve the accuracy of CodeBERT by up to 12.92% compared to the DNN model training without data augmentation. However, the syntax-preserved method code Refactoring performs surprisingly worse.

To the best of our knowledge, this is the first work that adapts data augmentation methods from NLP and empirically studies the effectiveness of incorporating data augmentation into training code models. Via the large-scale experiments, we found that data augmentation methods from NLP outperform the existing simple code refactoring data augmentation methods in most cases, for example, in clone detection-BigCloneBench, *Back-translation* outperforms code refactoring method in 3 out of 4 cases. Even though some data augmentation methods (e.g., *Random Swap*) can produce training data that slightly break the syntax of the source code, they are still useful in improving the quality of training in source code learning. Besides, when training data are scarce, data augmentation is especially important since it can significantly improve the quality of trained code models.

To summarize, our main contributions are:

- This is the first work that adapts data augmentation methods from NLP and empirically studies the effectiveness of incorporating data augmentation into training code models.
- Based on our empirical study, we have multiple findings that can help developers choose the best data augmentation methods to build their code models. One notable result is that the methods that slightly break the syntax rules of the source code are still helpful to model training in source code learning.

2. Background

2.1 Source code learning

In a nutshell, *source code learning* consists in learning the information from source code and using the learned information to solve the downstream tasks, such as code clone detection [2] and bug detection [3].

As mentioned in Section 1, code representation is a crucial technique that converts source code into a DNN-readable format to learn the features [10]. In this paper, we consider the widely-used code representation, namely *sequential representation* [1]. **Sequential representation** transforms the source code into a sequence of tokens (in a similar way to handling text data), where a token is the basic component of the code, such as a separator, an operator, a reserved word, a constant, and an identifier. In this way, the original source code is processed to a number of tokens, e.g., "`def func(a, b)`" is transformed to "`['def', 'func', '(', 'a', 'b', ')']`". Sequential representation keeps the context of the source code, which is useful for learning the syntactic information of source code.

2.2 Data augmentation in source code learning

Despite the great advantages of DNN, there are two main bottlenecks that prevent DNNs from achieving high performance, 1) the lack of high-quality labeled training data and 2) the different data distribution between training data and testing data. One simple solution to these two problems is to increase the size and diversity of training data. Data augmentation is proposed to automatically produce additional synthetic training data by modifying existing data without further human effort [11]. Generally, data augmentation involves a family of well-designed data transformation methods. For instance, in image processing, commonly used data augmentation methods include re-scaling, zooming, random rotating, padding, and adding noise.

Recently, software engineering researchers also considered data augmentation in source code learning [12], [13], and the proposed methods are known as *code refactoring*.

In general, code refactoring, originally used for code simplification, involves a family of techniques that rewrite the syntactic structure of source code while keeping the semantic information [14]. Commonly-used code refactoring techniques include *local variable renaming*, *duplication*, *dead store*, etc. For instance, *local variable renaming* is a method that changes the names of a code element, including symbols, files, directories, packages, and modules. Technically, this method modifies the source code slightly but does not change the semantic behavior of the program.

However, existing studies [9], [15] have shown that these simple strategies have limited advantages in improving the performance of code models. In this study, inspired by the analogy of source code to texts (as mentioned in Section 2-A), we empirically investigate whether data augmentation methods from NLP (that handles text data) can effectively enrich the diversity of training data for source code learning.

3. Adapting Data Augmentation Methods for Source Code Learning

As introduced in Section 2-A, source code can be represented as sequential data. Motivated by the similarity between source code representation (sequential tokens) and NLP data, we explore the adaptation of NLP-inspired data augmentation methods for source code learning.

Specifically, we follow recent survey papers [16] to identify seven data augmentation methods that are applicable to source code, as shown in Fig. 1. These methods are classified into three categories as follows:

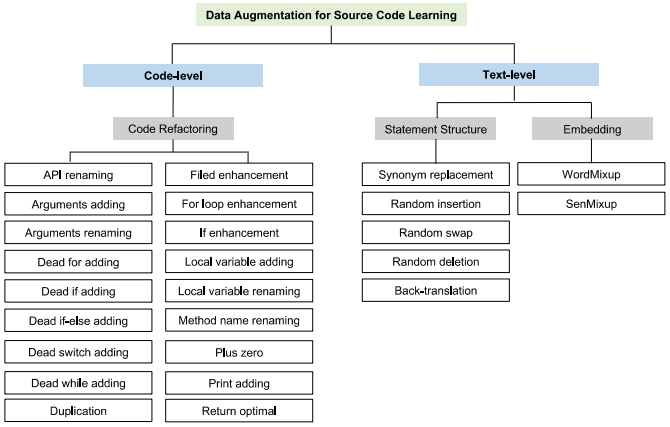


Figure 1. Data augmentation methods

- **Paraphrasing** can express the same information as the original form and has been commonly used in NLP [16]. In this paper, we select the *Back-translation* [17] method.
- **Noising-based methods** slightly add noise to the original data but keep their semantic information [18]. Four types of noise injection methods are employed in this study, namely, *synonym replacement*, *random insertion*, *random swap*, and *random deletion*.
- **Sampling-based methods** generate new synthetic data by linearly mixing the latent embeddings instead of directly operating on the raw text data. Unlike other methods, sampling-based methods are task-specific and require both data and label formats [16]. In this paper, we select two advanced sampling-based data augmentation methods used in NLP, namely *WordMixup* and *SenMixup* [19].

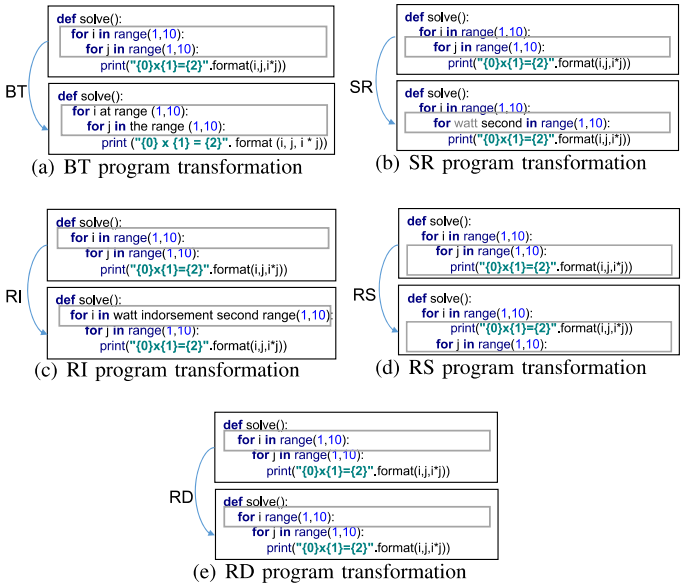
In the following, we elaborate on these seven data augmentation methods and particularly highlight the adaptations we have made in order to handle source code data.

3.1 Paraphrasing

Back-translation (BT) This method translates the original text into another language and then translates it back to the original one to generate additional data. In source code learning, we implement *BT* by applying the English-French translation model bidirectionally for each statement in a program. For example, in Fig. 2(a), after *BT*, we replace the statement "`in range`" in the original code with "`at range`" and "`in the range`" respectively. Note that these alterations may slightly break the syntax of the code data, nevertheless, they are rather minor, and the original relation between the features and the label is still preserved in the code data.

3.2 Noising-based methods

Synonym Replacement (SR) In NLP, this method randomly selects n words from a sentence and then replaces the selected words with one of its randomly chosen synonyms. Different from *BT*, to further enrich the diversity, *SR* usually refrains from substituting strings that are semantically similar to the original text data. Specifically, in source code learning, we



(a) BT program transformation (b) SR program transformation (c) RI program transformation (d) RS program transformation (e) RD program transformation

Figure 2. Examples of data augmentation methods from NLP to source code learning, with a code snippet from Python800-p00000-s024467653.py (For each sub-figure, the upper part shows the code without data augmentation, and the lower part shows the code after applying data augmentation.)

first randomly select n statements from a program. Then, each of the n words is replaced with one of its synonyms that is selected at random. In Fig. 2(b), we randomly select one statement from a program and then replace it with another string that is generated from its synonyms chosen at random. Similar to the case of *BT*, this method also preserves the original relation between the features and the label in the code.

Random Insertion (RI) In NLP, this method randomly inserts a random synonym of a random word into a sentence to generate augmented text data. Different from *RI* used in text data, we first select a random synonym of a random word in the chosen statement, then randomly insert this selected synonym into a random position of this statement. Generally, this process is repeated n times. In Fig. 2(c), we randomly insert the string that is generated from synonyms in a random position of the selected statement from the original code, i.e., "`for i in range(1,10)`". Again, this method is able to preserve the original relation between the features and the label in the code.

Random Swap (RS) In NLP, *RS* randomly chooses two words in a sentence and then swaps their positions. Although the semantics of text data is, in general, sensitive to the order of words, within a limited level of word swapping, the text after *RS* is often still understandable to humans. Therefore, *RS* can be used to produce augmented text data. In source code learning, we randomly select two statements of a program and swap their positions, and this process is usually repeated n times. In Fig. 2(d), we randomly select two statement "`for j in range(1,10)`" and

"`print("{0}x{1}={2}".format(i,j,i*j))`", and swap their positions. Despite the minor alteration of the order of the selected statements, this method can still preserve the original relation between features and the label in the code.

Random Deletion (RD) This method randomly removes some words in a sentence or some sentences in a document, with a probability p, to generate augmented text data. In source code learning, we randomly delete words in a randomly chosen statement. In Fig. 2(e), we delete words in a statement with probability $p = 0.01$. As a result, the operator "`in`" is removed from the statement "`for i in range(1,10);`" after *RD*. Similarly, this method can preserve the original relation between the features and the label in the code.

3.3 Sampling-based methods

We introduce two data augmentation methods based on *Mixup* [20], a popular data augmentation approach in *computer vision*. In that context, *Mixup* synthesizes new image data and their labels by linearly mixing the image features and the labels of two selected images. It has inspired the development of many data augmentation methods in other fields, including *WordMixup* and *SenMixup* in NLP, introduced as follows.

WordMixup and SenMixup. Originally, *WordMixup* interpolates the samples in the word embedding space, and *SenMixup* interpolates the hidden states of sentence representations [19]. We slightly modify *WordMixup* and *SenMixup* to adapt to source code learning. As shown in Eq. (1), there are two variants of *Mixup* in our study. The first one, denoted as *WordMixup*, interpolates samples in the embedding space of statement representation, and the second one, denoted as *SenMixup*, conducts the interpolation after a linear transformation and before it is passed to a standard classifier that generates the predictive distribution over different labels. Given two pairs $\left(x^i, y^i\right)$ and $\left(x^j, y^j\right)$, where x^i and x^j represent the code data, and y^i and y^j are their corresponding labels, the interpolated new data are obtained via *WordMixup* and *SenMixup*, as follows:

$$\begin{aligned} x^{ij}_{WordMix} &= \lambda x^i + (1-\lambda) x^j \\ x^{ij}_{SenMix} &= \lambda f(x^i) + (1-\lambda) f\left(x^j\right) \\ y^{ij}_{WordMix} &= \lambda y^i + (1-\lambda)\, y^j \\ y^{ij}_{SenMix} &= \lambda y^i + (1-\lambda)\, y^j \end{aligned} \tag{1}$$

Here *SenMixup* follows a similar workflow with [19], and $f(\cdot)$ denotes a linear transformation method that is able to ensure that the input and the output have the same dimension. Moreover, $x^{ij}_{WordMix}$ and x^{ij}_{SenMix} represent the new synthetic training data obtained by *WordMixup* and *SenMixup* respectively, and $y^{ij}_{WordMix}$ and y^{ij}_{SenMix} are their labels. The parameter λ denotes the *Mixup* ratio, and according to [20], it is sampled from a *Beta* distribution with a shape parameter α ($\lambda \sim Beta(\alpha, \alpha)$).

To better understand how the linear interpolation method works, we use an example to show the details of linearly mixing two different programs (Program A and B from Python800 with label 0 and label 1, respectively) as depicted in Fig. 3. First, we map a pair of programs (Data A and Data B) into the vector space through CodeBERT [6] and transform their labels into one-hot vectors with 800 classes. Next, we linearly mix the code vectors and label vectors, respectively, of Data A and Data B as the augmented training data (Data C) that could be used to train the model.

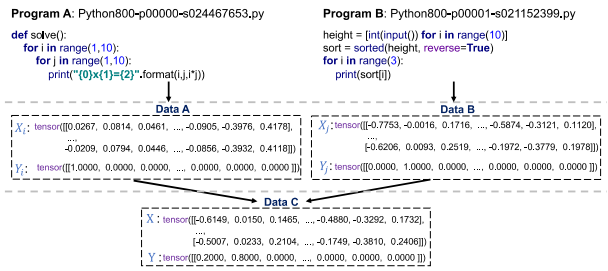


Figure 3. An example of linear interpolation of two programs

4. Study Design

In order to assess the effectiveness of the data augmentation methods in Section 3 in source code classification, we design three research questions, as follows:

RQ1: Can existing data augmentation methods produce accurate code models? Accuracy is the basic metric to evaluate the performance of a trained model. Therefore, we first assess whether data augmentation methods from NLP can improve the accuracy of code models. These methods are compared to training without data augmentation and training with code refactoring method (as introduced in Section 2-B). In this RQ, first, we prepare the original training data and randomly initialize code models, and then train these models using different data augmentation methods as listed in Fig. 1. Specifically, we compare these methods in terms of two metrics, namely, 1) the convergence speed of the model and 2) the final accuracy of the model with fixed training epochs.

RQ2: How does data volume affect the effectiveness of data augmentation methods? Given the original goal of data augmentation to address the issue of limited labeled data, it is essential to explore the effectiveness of data augmentation methods in a practical scenario with insufficient training data. To this end, we reduce the size of the training set and repeat the evaluation as in RQ1.

5. Experimental Setup

5.1 Baseline data augmentation

For comparison, we totally collect 18 code refactoring methods from the existing literature [12], [21], [22] as baselines, such as *local variable renaming*, *if loop enhance*, and *argument adding* in Fig. 1. For each code data, we randomly select one of these 18 code refactoring methods and apply it to the original code data to generate augmented code data.

TABLE I
DETAILS OF TASKS, DATASETS, AND DNN MODELS. #TRAINING: NUMBER OF TRAINING DATA. #TEST: NUMBER OF TEST DATA.

Dataset	Language	Task	#Training	#Test	Model
Java250	Java	Problem classification	48,000	15,000	
Python800	Python	Problem classification	153,600	48,000	BagofToken
CodRep1	Java	Bug detection	6,944	772	SeqofToken
Refactory	Python	Bug detection	3,380	423	CodeBERT
GCJ	Python	Authorship attribution	528	132	GraphCodeBERT
BigCloneBench	Java	Clone detection	90,102	4,000	

5.2 Tasks and datasets

Code classification servers for estimating programs' functionality automatically, which is crucial for software reuse [4]. Therefore, we first focus on code classification. Table I presents the dataset and model details. For all datasets, we directly follow the settings provided by the official projects to split the data into training, validation, and test sets.

Problem classification is a typical source code learning task that classifies the target functions of source code. Given multiple problems with detailed descriptions and the corresponding candidate source code, the trained model will identify the problem that the code is trying to solve. Two recently released datasets, Java250 and Python800 [4], are used. Java250 and Python800 have 250 and 800 problems, respectively,

Bug detection is to determine whether a piece of code contains bugs, which is most often considered as a binary classification problem. Refactory [3] and CodRep1 [23], two real-world datasets designed for bug repair, are used in our study.

Authorship attribution involves identifying the writer of a given code fragment by inferring the characteristics of programmers from their published source code, which is crucial for granting credit for a programmer's contribution and is also helpful for detecting plagiarism. We use the dataset from Google Code Jam (GCJ) provided by Yang *et al.* [24].

Clone detection aims to check whether a code pair is semantically identical or not, which helps prevent bug propagation and makes software maintenance easier. We use the widely used clone detection benchmark dataset BigCloneBench [2]. Note that Mixup and its variants are not suitable for this task since its input is code pairs.

5.3 Models

There are two paradigms for code learning, 1) using task-specific programming language (PL) models and 2) using pre-trained PL models.

- *Code learning with task-specific PL models.* This is a simple type of code learning where a code model is initialized randomly for a specific task and is trained using a task-related dataset from scratch. Generally, the trained models are lighter than the models using pre-trained PL models (e.g., 103 MB for BagofToken models vs. 487,737 MB for GraphCodeBERT models, as reported in our experiment) and can be deployed in machines with low computation resources.
- *Code learning with pre-trained PL models.* Different from task-specific models, pre-trained models are trained on a broad set of unlabeled data and can be used for a wide range of downstream tasks with minimal fine-tuning. Due to its large input volume, pre-trained models usually have better accuracy and higher generalization ability [25]. First, pre-trained PL embedding models are trained using multi-language datasets, e.g., Java, C++, and Python. Then, given a dataset that targets a specific downstream task, such as code clone detection, we fine-tune the pre-trained model accordingly and produce the final model.

We prepare four types of DNN models for each dataset, including pre-trained PL models and models trained from scratch.

- *Models trained from scratch.* Two types of models that need to be trained from scratch are studied, FNN (BagofToken) [4] and CNN (SeqofToken) [4]. FNN (BagofToken) is a basic mode type that only contains dense layers. CNN (SeqofToken) consists of both dense layers and convolutional layers.
- *Pre-trained models.* Besides, following the existing work [24], two well-known pre-trained models, CodeBERT [6] and GraphCodeBERT [5], are also considered in our study. CodeBERT is a bimodal model trained by using data from multiple programming languages, such as C, C++, Java, and natural languages. It follows the same spirit as BERT and treats programs as sequences during pre-training. To consider the semantic-level structure of programs, GraphCodeBERT adds data-flow information to the training data that can produce a more precise code representation. Notably, for pre-trained models, we fine-tune all the layers (including the encoder and decoder) in the models for downstream tasks.

5.4 Evaluation metrics

We evaluate the performance of trained DNN models from *clean accuracy*, which is a commonly used indicator for the quality of DNN models, measured by the following metric:

Accuracy is a basic metric that calculates the % of correctly classified data over the entire test set. Clean accuracy means the accuracy of models on the original test data. These data generally follow the same data distribution as the training data.

5.5 Implementation

The implementation of all the data augmentation methods is based on pure Python and the Numpy package, which makes it easy to extend this study to support more techniques in the future. Moreover, we also provide a code refactoring generator, including 18 different code refactoring methods that support both Java and Python languages. The models, including BagOfToken and SeqOfToken, are built using TensorFlow2.3 and Keras2.4.3 frameworks. CodeBERT and GraphCodeBERT are built using PyTorch1.6.0. We set the training epoch to 50 for the above four models. For the Mixup ratio that is set in augmenting training data, $\alpha = 0.1$ is our default setting. To alleviate overfitting, we adopt early stopping with patience 20. To lessen the impact of randomness, we train each model five times and report the average results with standard deviation.

TABLE II
ACCURACY ↑ (AVERAGE ± STANDARD DEVIATION, %) OF BAGOFTOKEN, SEQOFTOKEN, CODEBERT AND GRAPHCODEBERT. **NO AUG (BASELINE)**: WITHOUT DATA AUGMENTATION. BEST RESULTS ARE HIGHLIGHTED IN GRAY.

Model	DA method	Java250	Refactory	GCJ	BigCloneBench
BagofToken	No Aug	71.24 ± 0.04	85.15 ± 0.12	27.82 ± 0.14	85.23± 0.34
	WordMixup	73.12 ± 0.01	86.46 ± 0.28	28.43 ± 0.25	-
	SenMixup	75.33 ± 0.02	85.42 ± 0.31	29.23 ± 0.22	-
	Refactor	68.95 ± 0.03	85.03 ± 0.14	26.43 ± 0.12	85.54± 0.42
	SR	70.06 ± 0.05	81.91 ± 0.23	27.23 ± 0.21	85.45± 0.39
	RI	71.63 ± 0.03	85.81 ± 0.12	27.83 ± 0.33	86.28± 0.51
	RS	71.77 ± 0.02	86.33 ± 0.11	28.12 ± 0.24	86.48± 0.31
	RD	71.13 ± 0.05	85.68 ± 0.12	28.39 ± 0.26	86.87± 0.44
	BT	70.86 ± 0.02	73.96 ± 0.16	25.98 ± 0.21	85.65± 0.36
SeqofToken	No Aug	86.61 ± 0.05	85.55 ± 0.13	38.67 ± 0.12	90.69 ± 0.25
	WordMixup	94.42 ± 0.02	87.51 ± 0.22	38.98 ± 0.36	-
	SenMixup	95.35 ± 0.12	90.02 ± 0.27	39.34 ± 0.31	-
	Refactor	93.19 ± 0.28	85.49 ± 0.16	38.04 ± 0.24	91.14 ± 0.39
	SR	93.33 ± 0.35	81.64 ± 0.13	38.11 ± 0.21	91.23 ± 0.41
	RI	94.49 ± 0.16	87.11 ± 0.22	38.54 ± 0.33	91.09 ± 0.36
	RS	93.47 ± 0.22	81.81 ± 0.24	38.87 ± 0.23	92.67 ± 0.31
	RD	94.25 ± 0.31	84.64 ± 0.13	38.75 ± 0.21	92.34 ± 0.29
	BT	93.81 ± 0.25	81.38 ± 0.32	36.56 ± 0.22	90.86 ± 0.54
CodeBERT	No Aug	96.39 ± 0.03	96.22 ± 0.11	90.98 ± 0.14	96.89 ± 0.19
	WordMixup	96.31 ± 0.04	96.16 ± 0.12	91.34 ± 0.24	-
	SenMixup	96.56 ± 0.02	96.99 ± 0.24	92.16 ± 0.29	-
	Refactor	96.42 ± 0.05	95.94 ± 0.12	91.73 ± 0.17	96.95 ± 0.29
	SR	96.33 ± 0.02	96.69 ± 0.14	70.68 ± 0.08	96.98 ± 0.17
	RI	96.31 ± 0.06	96.45 ± 0.12	59.89 ± 0.34	97.31 ± 0.31
	RS	96.47 ± 0.04	96.71 ± 0.13	93.23 ± 0.09	96.99 ± 0.18
	RD	96.58 ± 0.03	96.45 ± 0.15	76.99 ± 0.11	97.01 ± 0.37
	BT	96.21 ± 0.02	94.33 ± 0.21	82.71 ± 0.12	97.02 ± 0.19
GraphCodeBERT	No Aug	96.47 ± 0.13	96.82 ± 0.14	93.98 ± 0.12	96.85 ± 0.15
	WordMixup	96.23 ± 0.05	96.18 ± 0.24	93.67 ± 0.23	-
	SenMixup	96.52 ± 0.09	96.46 ± 0.21	94.51 ± 0.09	-
	Refactor	96.56 ± 0.12	95.51 ± 0.26	91.73 ± 0.14	97.08 ± 0.21
	SR	96.54 ± 0.11	95.54 ± 0.28	89.47 ± 0.19	97.09 ± 0.16
	RI	96.58 ± 0.04	94.56 ± 0.31	80.45 ± 0.23	97.32 ± 0.36
	RS	96.49 ± 0.02	97.91 ± 0.19	94.74 ± 0.12	97.52 ± 0.11
	RD	96.69 ± 0.21	96.51 ± 0.18	91.73 ± 0.21	97.18 ± 0.28
	BT	96.55 ± 0.18	95.98 ± 0.32	80.45 ± 0.24	97.12 ± 0.18

We conduct all experiments on a server with 4 GPUs of NVIDIA RTX A6000.

6. EVALUATION RESULTS

Due to page limitations, we report the results of one dataset per task, specifically, Java250 for problem classification, Refactory for bug detection, GCJ for authorship attribution, and BigCloneBEnch for clone detection.

6.1 RQ1: Can existing data augmentation methods produce accurate code models?

As shown in Table II, *SenMixup* achieves the best performance in 6/12 cases. Surprisingly, in most cases (8/16), the code refactoring method can not improve the accuracy of models compared to *No Aug*. For SeqofToken models, again, *SenMixup* outperforms *No Aug* with accuracy improvements by up to 8.74% and 4.63% on average, and *Refactor* with accuracy improvements by up to 4.53% and 2.66% on average. The results recommend choosing *SenMixup* for traditional DNN models (e.g., FNN, CNN) to solve code classification tasks.

Moving to pre-trained PL models, first of all, we observe that compared to the results of the above two types of models, *SenMixup* is not sufficient to improve the accuracy of pre-trained models (only a maximum of 1.18% accuracy improvement). By contrast, data augmentation methods that slightly break the syntax of source code, such as *RS*, are more effective and have a clear improvement (by up to 2.25%) compared to *No Aug*, and (by up to 3.01%) compared to *Refactor*. In GraphCodeBERT, *RS* achieves the best accuracy improvement in three (out of four) datasets. However, even though *SenMixup* and *RS* achieve relatively better results than other methods, our statistical analysis *T-Test* and *Wilcoxon signed-rank test* found that their advantage is not significant; specifically, only in two out of 28 cases, the p-values are less than 0.05. This finding indicates that proposing a more powerful data augmentation method for code learning is a promising direction.

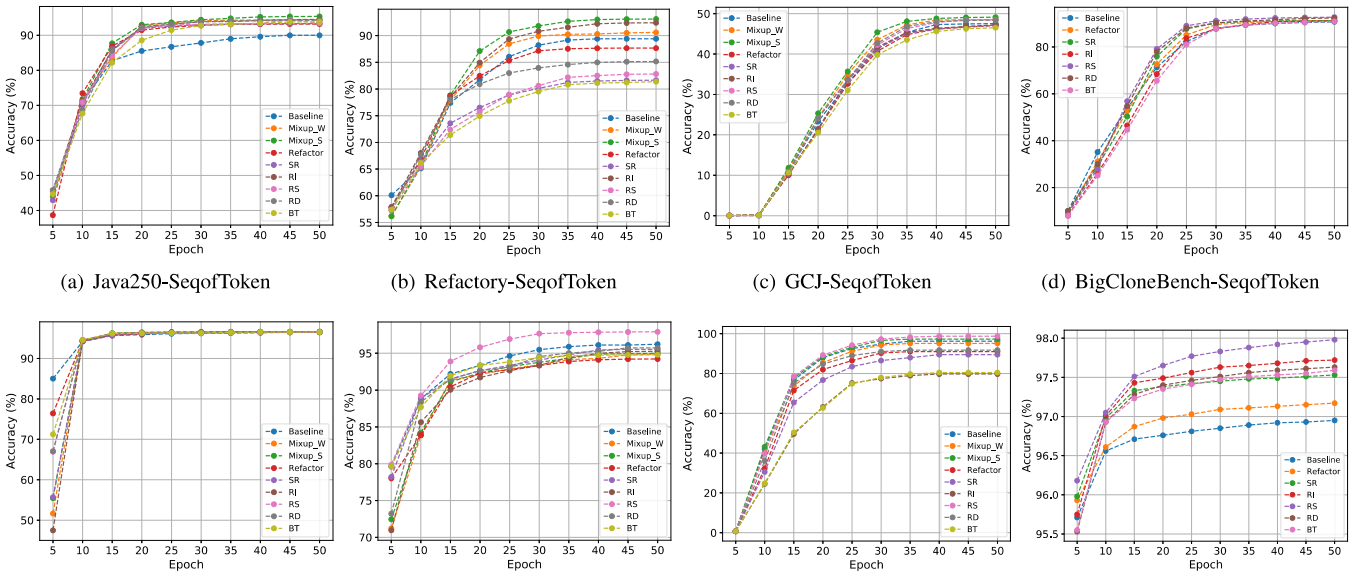
On the other hand, we investigate the convergence speed of models using different data augmentation methods. Fig. 4 depicts the results of training logs. From the results, we find that similar to findings that come from analyzing the final accuracy of models, there are two methods that have clearly better performance than others, *SenMixup* and *RS*. This indicates that, with limited computation budgets, these two methods are also recommended for practical use.

> **Answer to RQ1**: Data augmentation methods that linearly mix the code embedding, especially *SenMixup*, are effective in enhancing model performance. Surprisingly, the data augmentation method that slightly breaks the syntax rules achieves the highest accuracy boost in pre-trained PL models.

6.2 RQ2: How does data volume affect the effectiveness of data augmentation methods?

Data augmentation methods are used to enrich the size and also the diversity of training data, especially in the shortage of labeled training data. Therefore, it is necessary to explore whether data augmentation methods can be still effective in improving the accuracy of DNN models when the size of training data is rather small. In this RQ, we keep only 10%, 5%, 3%, and 1% of training data (for the authorship attribution task, we keep 10% and 50% of training data since the size of original training data is relatively small) and repeat the experiments conducted in Section 6-A. We choose two mainstream pre-trained PL models for our study. Here, we only report the results of CodeBERT models.

Table III and the left part of Table IV present the results of clean accuracy. For problem classification tasks (Java250), we can see that only in one case, CodeBERT (1%), data augmentation can improve the accuracy significantly (by up to 5.64% accuracy improvement). A similar phenomenon also happens in the clone detection task, the accuracy improvements produced by data augmentation are negligible. In contrast, for the bug detection task, there is a method (*SenMixup*) that can always significantly improve the accuracy of models with a margin from 3.32% to 12.92%. Interestingly, two noising-based methods, *SR* and *RI*, that perform well when using the entire training data fail to produce accurate models after reducing the size of the training data. For example, in Java250-CodeBERT (1%), *SR*- and *RI*-produced models only have 37.06% and 27.03% accuracy, while the baseline method (*No Aug*) has 49.62%. In conclusion, *SenMixup* is still the recommended method that has relatively better results than



(a) Java250-SeqofToken (b) Refactory-SeqofToken (c) GCJ-SeqofToken (d) BigCloneBench-SeqofToken

(e) Java250-GraphCodeBERT (f) Refactory-GraphCodeBERT (g) GCJ-GraphCodeBERT (h) BigCloneBench-GraphCodeBERT

Figure 4. Training log of models (SeqofToken and GraphCodeBERT) in Java250, Refactory, GCJ, and BigCloneBench.

TABLE III
ACCURACY ↑ (AVERAGE ± STANDARD DEVIATION, %) OF CODEBERT USING #% TRAINING DATA. **NO AUG (BASELINE)**: WITHOUT DATA AUGMENTATION. BEST RESULTS ARE HIGHLIGHTED IN GRAY.

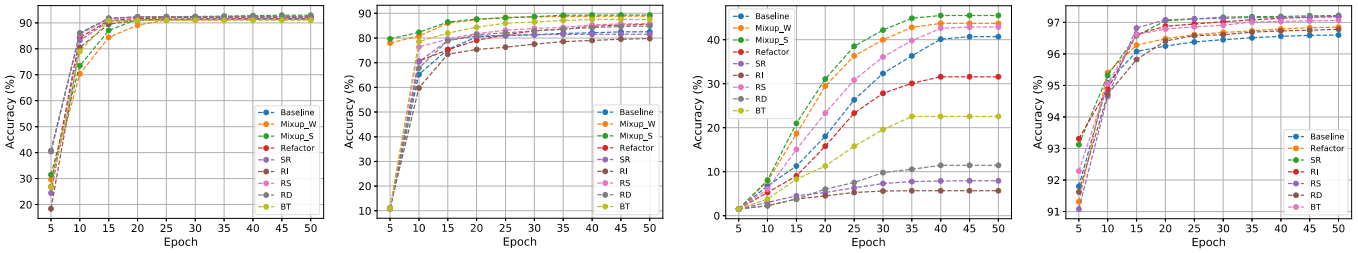
Model	DA method	Java250	Refactory	BigCloneBench
CodeBERT (10%)	No Aug	92.28 ± 0.03	82.51 ± 0.18	96.58 ± 0.15
	WordMix	91.33 ± 0.04	83.89 ± 0.13	-
	SenMixup	92.52 ± 0.07	85.83 ± 0.12	-
	Refactor	92.36 ± 0.05	85.58 ± 0.17	96.72 ± 0.33
	SR	92.09 ± 0.06	81.81 ± 0.13	96.89 ± 0.27
	RI	91.91 ± 0.02	79.67 ± 0.15	96.81 ± 0.19
	RS	92.39 ± 0.06	85.82 ± 0.19	97.06 ± 0.18
	RD	92.41 ± 0.09	84.87 ± 0.14	96.66 ± 0.21
	BT	91.01 ± 0.02	87.23 ± 0.16	96.93 ± 0.15
CodeBERT (5%)	No Aug	88.37 ± 0.13	81.32 ± 0.26	79.24 ± 0.51
	WordMix	88.52 ± 0.27	87.43 ± 0.25	-
	SenMixup	88.14 ± 0.18	88.89 ± 0.28	-
	Refactor	87.94 ± 0.15	76.83 ± 0.35	78.99 ± 0.55
	SR	86.14 ± 0.12	84.41 ± 0.24	78.58 ± 0.69
	RI	85.77 ± 0.14	79.91 ± 0.43	79.02 ± 0.25
	RS	88.55 ± 0.16	87.23 ± 0.27	80.02 ± 0.65
	RD	88.75 ± 0.15	77.31 ± 0.16	80.06 ± 0.86
	BT	85.18 ± 0.13	90.07 ± 0.22	80.04 ± 0.45
CodeBERT (3%)	No Aug	78.15 ± 0.13	75.89 ± 0.23	78.67 ± 0.44
	WordMix	78.89 ± 0.25	87.33 ± 0.32	-
	SenMixup	79.19 ± 0.12	88.81 ± 0.43	-
	Refactor	74.62 ± 0.15	86.29 ± 0.36	79.53 ± 0.89
	SR	75.03 ± 0.12	84.41 ± 0.39	79.54 ± 0.67
	RI	73.05 ± 0.14	87.71 ± 0.44	79.39 ± 0.39
	RS	77.59 ± 0.11	81.81 ± 0.31	79.77 ± 0.22
	RD	78.18 ± 0.12	76.12 ± 0.24	79.19 ± 0.39
	BT	73.01 ± 0.16	88.65 ± 0.33	79.67 ± 0.37
CodeBERT (1%)	No Aug	49.62 ± 0.14	76.83 ± 0.21	72.02 ± 0.65
	WordMix	53.21 ± 0.26	87.11 ± 0.44	-
	SenMixup	55.26 ± 0.28	88.12 ± 0.41	-
	Refactor	34.59 ± 0.26	72.11 ± 0.53	72.03 ± 0.69
	SR	37.06 ± 0.35	82.74 ± 0.43	73.43 ± 0.88
	RI	27.03 ± 0.49	88.18 ± 0.54	72.67 ± 0.16
	RS	49.53 ± 0.38	82.98 ± 0.52	72.51 ± 0.38
	RD	49.78 ± 0.33	76.84 ± 0.44	72.47 ± 0.89
	BT	36.68 ± 0.28	88.42 ± 0.55	72.55 ± 0.59

TABLE IV
ACCURACY ↑ (AVERAGE ± STANDARD DEVIATION, %) OF CODEBERT USING #% TRAINING DATA, RESPECTIVELY, ON TEST DATA. **NO AUG (BASELINE)**: WITHOUT DATA AUGMENTATION. DATASET: GCJ.

DA method	Test accuracy	
	10%	50%
No Aug	40.69 ± 0.11	86.47 ± 0.15
WordMix	42.14 ± 2.01	84.21 ± 0.23
SenMixup	44.51 ± 1.31	86.98 ± 0.31
Refactor	31.58 ± 0.21	78.21 ± 0.17
SR	7.52 ± 0.15	57.89 ± 0.09
RI	3.76 ± 0.13	22.56 ± 0.15
RS	41.67 ± 0.17	86.98 ± 0.19
RD	12.78 ± 0.24	55.64 ± 0.14
BT	23.32 ± 0.27	69.17 ± 0.06

others even though its advantage is not significant (p-values are greater than 0.05) according to our statistical analysis.

Then, we check the convergence speed of models using different data augmentation methods. Fig. 5 represents the training logs of CodeBERT (10%) models. From the results, we can see that when data augmentation methods are used, the convergence speed of models is easily affected by a drop in the data scale. For instance, compared to the results of model training using the entire dataset, after 10 epochs, GCJ-CodeBERT has more significant accuracy improvement with data augmentation methods, i.e., *RS*, *SenMixup*, and *WordMixup*. Besides, we can find all data augmentation methods effectively improve the performance of accuracy in BigCloneBench-CodeBERT compared to *No Aug* after 20 epochs. This can be beneficial for us to select more effective data augmentation methods to reduce the time (computation budget) cost of models when the training data is in a very limited situation.



(a) Java250 (b) Refactory (c) GCJ (d) BigCloneBench

Figure 5. Training log of CodeBERT using 10% training data.

Answer to RQ2: When training data is scarce, *SenMixup* is the best choice which outperforms *No Aug* by up to 12.92% in CodeBERT in terms of accuracy. On the contrary, the syntax-preserved method *Refactor* performs surprisingly worse, e.g., in 8 (out of 14) cases, *Refactor* harms the accuracy of models compared to *No Aug*.

7. Discussions and Threats to Validity

7.1 Is data augmentation necessary for source code learning?

First, the most important question is whether it is necessary to use data augmentation when preparing code models. From our empirical study, the answer is yes. In the case of using a suitable data augmentation method, e.g., *SenMixup*, the trained models have higher accuracy (by up to 12.92%) than the models without using data augmentation. However, the results also demonstrate that when using pre-trained PL models, though data augmentation can still improve the performance of the model, the improvement is not significant compared to the case without pre-trained PL embeddings. The reason could be that, essentially, pre-training already plays the role of data augmentation that enhances the whole process of model training, and consequently, other data augmentation techniques become not as useful as they are in the cases without pre-training. An in-depth analysis of this phenomenon will be an interesting future research direction.

7.2 Is preserving syntax rules necessary in source code-based data augmentation?

The previous research has shown that, for natural language, although the semantics of text data is sensitive to their syntactic change, it could remain readable to humans [16] and valid as additional training data if the change happens within a limited range that does not break the original relations between the text data and their labels. Indeed, that is why noising-based data augmentation methods, such as *RI* and *RS* (see Section 3), are still very useful in NLP, as shown by the recent study [26]. In the context of source code learning, our experimental results suggest a similar conclusion, namely, even though some data augmentation methods can produce training data that slightly break the syntax of the source code, these data are still useful in improving the quality of training in source code learning. Indeed, as reported by our experiments, the pre-trained PL models using the *RS* method can achieve higher accuracy (by up to 14.94%) than the models using the baseline *Refactor* method.

7.3 Threats to Validity

The internal threat to validity comes from the implementation of standard training and data augmentation methods. The code of model training, *SenMixup*, and *WordMixup* methods are from the official projects. The code refactoring methods for the Java language come from the existing works [21], [22], and we adapt the implementation to the Python language.

The external threats to validity lie in the selected code-related tasks, datasets, DNNs, and data augmentation methods. We consider four different code-classification tasks, including problem classification, bug detection, authorship attribution, and clone detection in our study, a total of six datasets for the above tasks. Especially two popular programming languages in the software community (Java and Python) are included. We apply four types of DNN models, including two mainstream pre-trained PL models. For data augmentation methods from code, code refactoring methods cover the most common ones in the literature. Data augmentation methods of NLP come from the most classic method, which is comprehensively adopted from the number of citations of papers.

The construct threats to validity mainly come from the parameters, randomness, and evaluation measures. We follow the original recommendation of Mixup to set its parameters. The parameters of data augmentation methods from NLP also follow the original release. We repeat each experiment five times and report the average results to reduce the influence of randomness.

8. Related Work

We review related work about data augmentation for source code learning and empirical study on source code learning.

8.1 Data augmentation for source code learning

Data augmentation has achieved enormous success in the machine learning field [11]. Inspired by its success, recently researchers devoted considerable effort to leveraging the data augmentation technique in big code tasks to improve the performance of code models in terms of accuracy and robustness. Adversarial training [27], which produces a set of adversarial examples to the training data, has been studied as the data augmentation method in code learning. Zhang *et al.* [28]

proposed a code data augmentation method that employs the metropolis-Hastings modifier (MHM) algorithm [29] to improve the capability of deep comment generation models. Mi *et al.* [30] generated the additional data from Auxiliary Classifier generative adversarial networks (GANs). Besides, as a program transformation method that is specially designed for code, code refactoring has been used as a mainstream code data augmentation method. Yu *et al.* [9] designed program transformation rules for Java and evaluated the effectiveness of using these program transformations as code data augmentation in three big code-related tasks. Allamanis *et al.* [12] used four simple code rewrite rules as code data augmentation methods for improving the generalization of the code model. Compared to the above works, our study is the first one that assesses the effectiveness of two types of data augmentation methods for code learning, namely, source code and text data.

8.2 Empirical studies on source code learning

Recently, many works conducted empirical studies to explore the topic of ML4Code. Chirkova *et al.* [31] conducted a thorough empirical study to evaluate the capabilities of using Transformer to solve three downstream tasks related to code learning, including code completion, function naming, and bug fixing. Zhang *et al.* [32] empirically analyzed current testing and debugging practices for machine learning programs. They revealed that the interaction with the platform execution environments could easily cause machine learning program failures, moreover, current debugging is insufficient to locate the fault in machine learning code well. This work is useful for programmers to improve the quality of the source code of machine learning. Yan *et al.* [33] conducted a comprehensive empirical study on code search using machine techniques. Their empirical evaluation results revealed that machine learning techniques are more effective for queries on reusing code. More recently, Hu *et al.* [25] empirically studied the distribution shift problem of code learning and defined five types of shift for code data.

Different from the existing empirical studies, our work investigates data augmentation on source code classification that has rarely been studied to date.

9. Conclusions and Future Work

Our empirical study highlights the importance of data augmentation in code learning and provides insights into the effectiveness of various augmentation methods for different downstream tasks and DNN models. Specifically, linear interpolation methods such as *SenMixup* and *Manifold-Mixup* are found to be effective in improving the accuracy of most DNNs, except for pre-trained programming language (PL) models. Methods like random deletion (*RD*) and random swap (*RS*) that slightly break the syntax of source code are particularly effective for pre-trained PL models. Particularly, linear interpolation methods can handle the situation when the training data is limited. Moreover, in light of these findings, our study paves the way for further improving the effectiveness of data augmentation in code-related tasks, with the ultimate goal of enhancing program understanding and accelerating software development.

In future work, we plan to:

- Extend our study to explore how data augmentation methods affect non-classification code tasks, e.g., code summarization and code generation.
- Study how to use data augmentation methods to help fine-tune large language models (e.g., LLaMA) for code tasks.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, July 2018.

[2] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. USA: IEEE Computer Society, p. 476–480. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1109/ICSME.2014.77

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

[4] 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," 2021, arXiv:2105.12655.

[5] 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," 2020, arXiv:2009.08366.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: a pre-trained model for programming and natural languages," pp. 1536–1547, November 2020.

[7] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.

[8] D. Hendrycks and T. Dietterich, "Benchmarking neural network robustness to common corruptions and perturbations," *Proceedings of the International Conference on Learning Representations*, 2019.

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

[10] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" in *ICML*, 2021, pp. 8476–8486.

[11] 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.

[12] M. Allamanis, H. R. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *Advances in Neural Information Processing Systems*, 2021.

[13] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, p. 287–298. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3510003.3510062

[14] 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.

[15] 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, vol. 119. PMLR, July 2020, pp. 896–907.

[16] S. Y. Feng, V. Gangal, J. Wei, S. Chandar, S. Vosoughi, T. Mitamura, and E. Hovy, "A survey of data augmentation approaches for nlp," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Association for Computational Linguistics, August 2021, pp. 968–988.

[17] Q. Xie, Z. Dai, E. Hovy, M.-T. Luong, and Q. V. Le, "Unsupervised data augmentation for consistency training," in *NIPS'20*, 2020.

[18] J. Wei and K. Zou, "Eda: easy data augmentation techniques for boosting performance on text classification tasks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 6382–6388.

[19] H. Guo, Y. Mao, and R. Zhang, "Augmenting data with mixup for sentence classification: an empirical study," 2019, arXiv:1905.08941.

[20] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: beyond empirical risk minimization," in *International Conference on Learning Representations (ICLR)*, 2018.

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

[22] M. Wei, Y. Huang, J. Yang, J. Wang, and S. Wang, "Cocofuzzing: testing neural code models with coverage-guided fuzzing," *IEEE Transactions on Reliability*, 2022.

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

[24] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. Association for Computing Machinery, 2022, p. 1482–1493. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3510003.3510146

[25] Q. Hu, Y. Guo, X. Xie, M. Cordy, L. Ma, M. Papadakis, and Y. L. Traon, "Codes: towards code model generalization under distribution shift," in *ICSE: New Ideas and Emerging Results (NIER)*, 2023.

[26] V. Marivate and T. Sefara, "Improving short text classification through global augmentation methods," in *Machine Learning and Knowledge Extraction*, 2020, pp. 385–399.

[27] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations (ICLR)*, May 2015.

[28] 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*, 2021, p. 185–188.

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

[30] 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.

[31] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 703–715. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3468264.3468611

[32] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1159–1170. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3377811.3380362

[33] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for," *A benchmark and an empirical study on code search with natural-language queries*, 2020.