Data Augmentation for Boosting Source Code Learning

董, 沢銘

Data Augmentation for Boosting Source Code Learning

Zeming Dong

Graduate School of Information Science and Electrical Engineering

Kyushu University

A thesis submitted for the degree of

Doctor of Philosophy in Engineering

December 2023

Abstract

With the rise of deep code models, we are witnessing the success of code intelligence. However, current research mainly focuses on developing new code models or fine-tuning large pre-trained models for code-related tasks. One significant issue that often remains overlooked is the preparation of high-quality data to boost the training of these code models. In contrast, the importance of data quality and diversity via techniques namely data augmentation is well-established in fields such as *computer vision* and *natural language processing*. Surprisingly, this aspect has not been extensively explored in source code learning, and most existing practices of data augmentation in source code learning are limited to simple syntax-preserved methods such as *code refactoring* and are not sufficiently effective.

To address these challenges and contribute to the advancement of data augmentation in source code learning, this thesis primarily focuses on enriching effective and reliable code data augmentation methods. These include syntax-preserved methods and syntax-broken methods, and the work is divided into three phases:

- Essentially, source code is often represented sequentially as text data. Inspired by this analogy, we take an early step to investigate whether data augmentation methods originally for text data are effective for source code learning. Our focus centered on understanding tasks, and we conducted a comprehensive empirical study spanning four critical code-related problems and four *deep neural network (DNN)* architectures. Our findings identify that the data augmentation methods can produce more accurate and robust models for source code learning and show that the augmented data are still useful even if they slightly break the syntax of the source code.
- Sequential representation ignores the semantic information of the source code. To overcome this limitation, the graph-structural representation of the source code is proposed. In light of the graph nature of the source code, we propose to apply the data augmentation methods used for graph-structured data in graph learning to the tasks of source code learning. We conduct a comprehensive empirical study to evaluate whether such new ways of data augmentation are more effective than the existing simple code refactoring methods in terms of producing more accurate and robust models. Specifically, we evaluate four critical software engineering tasks and seven neural network architectures to assess the effectiveness of five data augmentation methods. Our findings identify that linear interpo-

lation can significantly improve both the accuracy and robustness of the trained models for source code learning.

- We introduce a data augmentation approach MixCode that draws inspiration from Mixup in computer vision. MixCode aims to effectively supplement valid training data without manually collecting or labeling new code. Specifically, we first employ code refactoring methods to create transformed code instances that maintain consistent labels with the original data. Subsequently, we adapt the Mixup to combine the original code with these transformed code samples, thereby augmenting the training data. Our evaluation of this approach includes two programming languages, two understanding tasks, four benchmark datasets, and seven different model architectures. Our Experimental results demonstrate that MixCode outperforms baseline data augmentation approaches in terms of accuracy and robustness.

In summary, this thesis fills the gaps in effective and reliable data augmentation for source code learning by (1) introducing text-oriented data augmentation for source code learning, (2) providing graph-oriented data augmentation for source code learning, and (3) developing MixCode, the first online code data augmentation method.

To my parents.

Acknowledgments

Throughout this thesis, I have received significant support and assistance, and I am grateful to many individuals.

First, I would like to thank Prof. Jianjun Zhao, who led me to software engineering and offered me the opportunity to pursue my doctoral studies under his supervision. He believed in my research and gave me valuable feedback, which has been instrumental to my academic journey.

Second, I am also thankful to all co-authors, including Dr. Zhenya Zhang, Dr. Yuejun Guo, Dr. Maxime Cordy, Prof. Mike Papadakis, and Prof. Yves Le Traon, for their guidance in conducting rigorous research and effective communication. The insightful discussions we had allowed me to become a better researcher. Of all the people supporting my thesis, I am especially grateful to Qiang Hu for his outstanding support.

Third, I would like to thank all the members of my Ph.D. defense committee. It is a great honor to have them serve on my defense committee, and I sincerely appreciate their efforts in reviewing my dissertation and evaluating my Ph.D. work.

I would like to express my sincere gratitude to Prof. Martin Monperrus and Prof. Benoit Baudry for providing me with the opportunity to delve deeper into the field of software engineering at KTH. I am also thankful to my former colleagues at KTH, Zimin, He, Long, Jian, and Zhongxing, who provided significant help to me both in life and study.

Last but most importantly, I would like to thank my parents for their financial support throughout my doctoral studies and for constantly encouraging me to complete my Ph.D. dissertation.

At the end of this journey, I would like to conclude with my favorite quote drawn from *Dr. William Smith Clark*.

Boys, be ambitious! —— Dr. William Smith Clark

Hokkaido University Library

Mutual encouragement for you and me!

Zeming Dong
Kyushu University, Japan
December 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"The performance of a machine learning (ML) model is upper bounded by the quality of the data."* —— **IBM Research** [1]

1.1 Motivation

The integration of *machine learning (ML)* techniques into software engineering has garnered increasing attention. ML models, including specifically designed models for code understanding, such as *CodeBERT* [19], GraphCodeBERT [25], and *CodeT5* [82], have demonstrated their effectiveness across various code-related tasks including code clone detection [88], program classification [62], and vulnerability detection [50]. ML already achieved significantly better results than traditional software engineering techniques. Moreover, the great success of large language models in software engineering [91] from the latest works pushes this trend further forward. Therefore, applying the power of ML to code-related tasks is becoming increasingly crucial.

Usually, building effective *deep neural networks (DNNs)* for code-related tasks involves several critical steps, from designing code-specific model architectures to preparing training data, model training, and testing. Researchers from the software engineering community have indeed contributed significantly to each of these aspects, including developing code-specific model architectures like *Code2Vec* [3], constructing code-related datasets such as *CodeXGLUE* [51], and exploring various training

[1] https://research.ibm.com/publications/overview-and-importance-of-data-quality-for-machine-learning-tasks

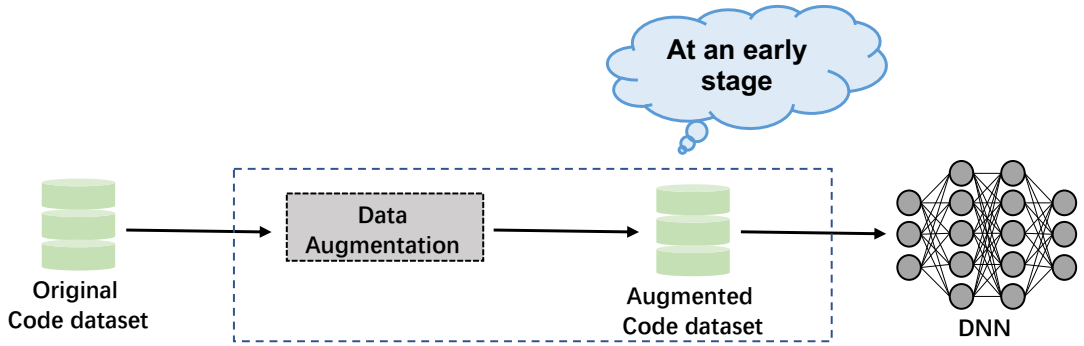


Figure 1.1: Overview of data augmentation for source code data

strategies, such as pre-training and fine-tuning for code models [19]. Additionally, recent efforts have extended the evaluation of code models to include not only accuracy but also robustness. However, the quality of training data, which serves as the foundation for code model training, has not received as much attention as it deserves. Recent research [79] has demonstrated that the quality of training data plays a crucial role in determining the performance of code models. This highlights the importance of data augmentation and preparation techniques in improving code model quality and, in turn, their performance in various software engineering tasks [72].

Indeed, preparing training data for code models is a labor-intensive process that involves two fundamental steps: *data collection* and *data labeling* [1]. While collecting raw code data from open-source repositories, like *GitHub* [2], has become relatively straightforward, data labeling, especially for code-related tasks, remains a complex and costly endeavor. This is primarily because it requires human effort, often from individuals with professional domain knowledge. Moreover, ensuring the prepared training data is of sufficient quality and quantity to train a code model effectively is a significant challenge. The accuracy and completeness of the labels, the diversity of code examples, and the representativeness of the data all contribute to the overall quality of the training dataset. Insufficient or poorly-labeled training data can severely impact the performance of a source code model. As a result, the field of training data preparation for code models is still under exploration and challenging [14].

Data augmentation is a widely adopted technique in fields such as *computer vision (CV)* [99] and *natural language processing (NLP)* [85]. It's used to increase the

[2] `https://github.com`

diversity and quality of training data by generating new samples based on existing labeled data. The key idea behind data augmentation is to apply various transformations to input samples, creating new samples while maintaining the same semantic content (i.e., the labels of the generated samples). In the field of software engineering, data augmentation shown in Figure 1.1 is a relatively unexplored area. While it has been extensively studied in other domains, its potential benefits and optimal methods for source code data remain an open problem. Recent research [15] has found that existing code augmentation methods have limited advantages for code models, highlighting the need for more effective and domain-specific approaches to data augmentation. The challenge in designing data augmentation techniques for source code data lies in maintaining the syntactic and semantic integrity of source code. Unlike text or image data, source code has specific syntax and structure that must be preserved during augmentation. This presents unique challenges and opportunities for developing code-specific data augmentation methods.

1.2 Contribution

A significant and valuable research focus is boosting DNN models for source code tasks through data augmentation. Given the unique challenges of source code data, developing effective specific data augmentation techniques to the requirements of source code models can significantly contribute to the field of code understanding and analysis. This doctoral thesis concentrates on this area and advances the understanding of how to improve the performance of deep learning models for source code analysis. The contribution of our work is as follows:

- **Text-Oriented Data Augmentation.** In source code learning, the 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 begin by surveying existing data augmentation methods in the literature. Subsequently, we categorize these methods to understand their applicability to source code learning. As a result, we identify seven data augmentation methods originally used in NLP but appear to have potential applications in source code learning. We evaluate the performance of these adapted data augmentation methods. Specifically, we assess their impact on improving the

accuracy and robustness of code models. 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.

This work has been published in the 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C 2023), and its extended version is currently under review in an international peer-reviewed journal.

- **Graph-Oriented Data Augmentation.** Source code can be effectively represented as graph data structures. Graph learning is a specialized approach designed for graph-structured data, often applying *graph neural networks (GNNs)* to process graph data. Considering the graph-like nature of source code, we propose applying data augmentation techniques from graph learning to source code learning tasks. This is the first study that applies graph data augmentation methods to source code learning and empirically examines the effectiveness of integrating data augmentation into the training of code models. Via large-scale experiments, we observed that data augmentation methods that are derived from graph learning outperform existing simple code refactoring methods in most cases. Besides, despite some graph data augmentation methods that can generate training data that mildly break the syntax of the source code, they still prove to be beneficial in enhancing the quality of training in source code learning.

  This work has been accepted by the Knowledge-Based Systems (KBS) in 2023.

- **MixCode.** We propose the first online data augmentation framework called MixCode, designed for source code learning. MixCode is developed to enhance the training process of DNN models. In a broad sense, MixCode follows a similar paradigm to Mixup [99], a well-known data augmentation technique initially created for image classification. Mixup linearly combines training data, including their labels, to increase the volume of training data. In source code learning,

MixCode involves two primary steps: firstly, it generates modified data using various code refactoring techniques, and subsequently, it linearly combines the original code with the transformed code to create new data samples. Experimental results demonstrate that MixCode outperforms all baseline data augmentation approaches in terms of accuracy and robustness. Besides, we also empirically show that the selection of refactoring methods and Mixup strategies are also important factors affecting the performance of MixCode.

This work has been published in the 30th International Conference on Software Analysis, Evolution and Reengineering (SANER 2023).

1.3 Thesis Structure

The rest of this thesis is organized as follows:

- Chapter 2 introduces background information about source code learning and related works in the field of deep learning-based software engineering.
- Chapter 3 describes the work of Text-Oriented data augmentation for source code models.
- Chapter 4 introduces the work that we adapt data augmentation from the field of graph learning to source code models.
- Chapter 5 introduces MixCode, the first online data augmentation for source code models.
- Chapter 6 concludes this thesis.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Graph Neural Network

Graph Neural Networks (GNNs) [22, 90] are a class of neural networks designed for analyzing and processing data with graph structures, such as text or source code. These networks leverage the inherent graph structure as an inductive bias to learn representations of the data. Typically, a GNN involves three main steps, including *Aggregate*, *Update*, and *Pooling*. GNN starts by computing the representation of a node within the graph. It does this by iteratively aggregating information from its adjacent nodes. The process begins with an initialized node representation. Next, GNN generates a new representation for each node. This new representation is created by combining the previously aggregated information with the node's existing representation. A non-linear neural network, like a *Gated Recurrent Unit (GRU)*, is often used to perform this update. Finally, the GNN creates a final representation of the entire graph data. This is achieved by sampling features from all the nodes in the graph. There are various types of GNN models, and their primary differences lie in how they handle the aggregation, update, and pooling steps. For instance, *Gated Graph Sequence Neural Networks (GGNNs) [49]* use GRU to update the new hidden state during information processing. These variations in GNN architectures allow them to learn and represent different graph-structured data types effectively.

Formally, given a graph data $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex (node) set, and $\mathcal{E}$ means

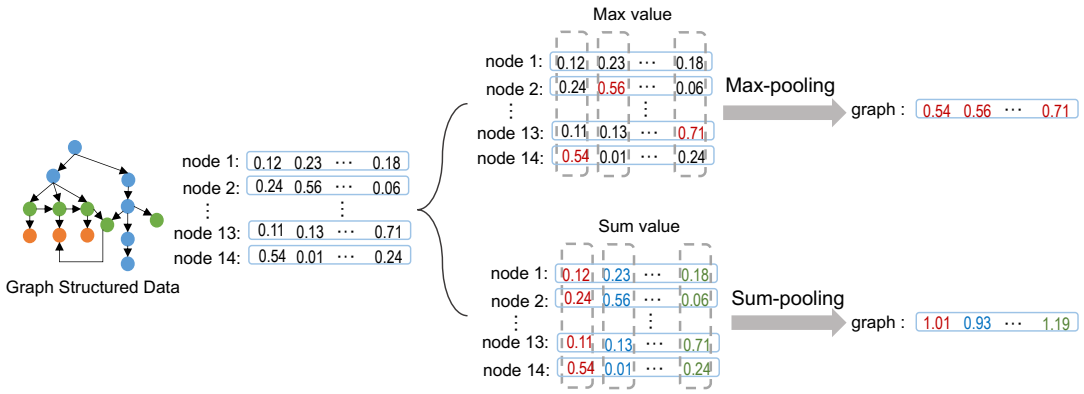


Figure 2.1: An example of graph Max-pooling and Sum-pooling

the edge set, we simply formulate the graph representation that is generated by *GNN* $x^{\mathcal{G}}$ as follows:

$$\begin{aligned} \mathbf{H}_u^k &= \mathbf{GNN}\left(\mathbf{H}_u^{k-1}, W, \sum_{v\in\mathcal{N}(u)} \mathbf{H}_v^{k-1}\right) \\ \mathbf{H}^{\mathcal{G}} &= \mathcal{P}(H,\mathcal{V}) \quad \text{s.t.} \quad H = \mathbf{H}_u^k \quad (u\in\mathcal{V}) \end{aligned} \tag{2.1}$$

where $\mathbf{H}_u^{k-1}$ and $\mathbf{H}_v^{k-1}$ denote the representation (shown as a matrix) of node u and v ($u,v\in\mathcal{V}$, $uv\in\varepsilon$) at the $(k-1)$-*th* iteration, W represents the trainable parameter value (shown as a matrix), $\mathcal{N}(u)$ represents the set of neighbor nodes of node u, and $\mathcal{P}$ signifies the operator responsible for sampling features from all nodes to create the final graph representation $\mathbf{H}^{\mathcal{G}}$, which is a matrix representation of the entire graph's vector $x^{\mathcal{G}}$. As shown in Eq. 2.1, firstly, a GNN neural network $\mathbf{GNN}(\cdot)$ is applied to iteratively update the feature of each node of a graph via the message passing aggregated from its' neighborhood $\sum_{v\in\mathcal{V}(u)}\mathbf{H}_v^{k-1}$ (**Alternating Layers**). Afterward, the sampling operator $\mathcal{P}(\cdot)$ is applied to generate the representation of the entire graph, capturing the global graph information after k steps of iteration (**Graph Pooling Layer**).

As depicted in Figure 2.1, the graph-structured data consists of 14 nodes, and the GNN's graph pooling layers generate the overall graph embedding. This is achieved by employing various fundamental sampling functions, such as Sum-pooling and Max-pooling, as discussed in the work [13].

Furthermore, our study explores seven models that necessitate training from the ground up. These models encompass *Graph Convolutional Networks (GCNs)* [42],

Graph Isomorphism Networks (GINs) [93], *Graph Attention Networks (GATs)* [75], *Gated Graph Sequence Neural Networks (GGNNs)*, and *GraphSAGE (SAGE)* [28]. A brief overview of each of these models is as follows:

GCN is a variant of convolutional neural networks tailored for processing graph-structured data. **GIN** generalizes the Weisfeiler-Lehman WL test, optimizing the discriminative capability of GNNs. **GCN-Virtual and GIN-Virtual** models introduce virtual nodes to augment the aggregation phase. They add an artificial node to each graph, connecting it bidirectionally to all graph nodes. **GAT** is a prominent GNN architecture that employs the attention mechanism during the graph message-passing process. **GGNN** is a GNN variant that updates the new hidden state using a Gated Recurrent Unit (GRU). **SAGE** is designed for inductive representation learning on large graphs, mapping each node into a low-dimensional vector space.

2.1.2 Source Code Learning

In a nutshell, source code learning involves using machine learning, particularly deep learning models, to understand and work with source code. This process begins with *code representation*, which transforms a set of programs into vectors or numerical representations. These vectors are then used to train machine learning models. The models learn from the information within these vectors and apply the acquired knowledge to address various downstream tasks, such as automated program repair [27], automated program synthesis [102], automated code comments generation [35], and code clone detection [80]. There are two paradigms for code learning, 1) using task-specific PL models and 2) using pre-trained *programming language (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 experiments). They 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

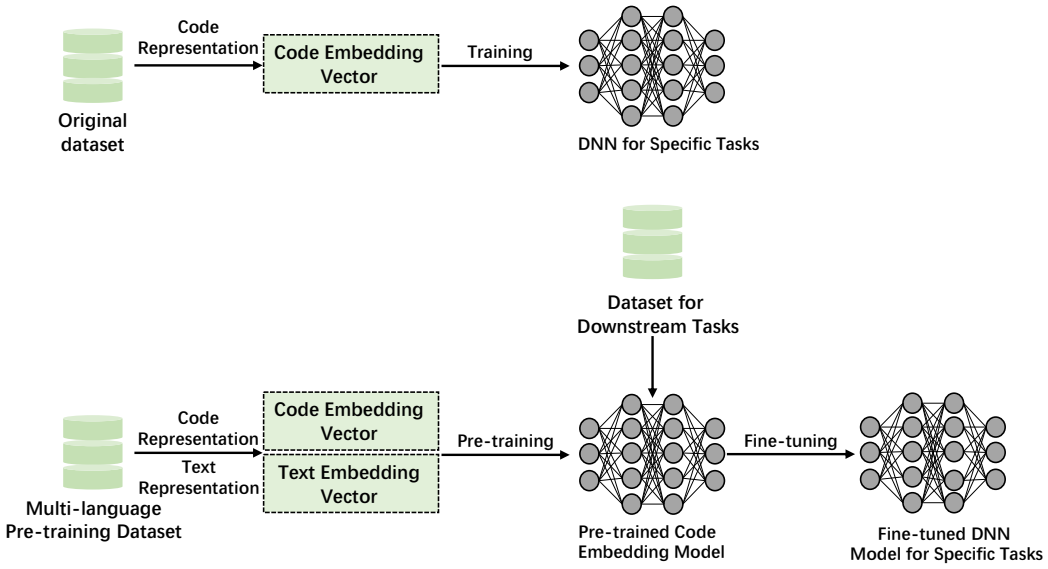


Figure 2.2: Workflow of source code learning (top: with a task-specific model, bottom: with a pre-trained model.)

large input volume, pre-trained models usually have better accuracy and higher generalization ability [34]. Fig. 2.2 shows the workflow (bottom) of this learning paradigm. First, pre-trained code 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.

As mentioned earlier, code representation is crucial to converting source code into a readable format for DNNs to learn features [59]. In our study, we consider two widely used code representation techniques, sequential representation and structural representation:

Sequential representation converts source code into a sequence of tokens (similar to handling text data). Each token represents an essential code component, such as separators, operators, reserved words, constants, and identifiers. This tokenization process enables the code to be processed and analyzed similarly to handling text data in *natural language processing (NLP)*. In this way, the source code can be transformed into numbers of tokens, e.g., "`def main()`" is converted to "`['def', 'main', '(', ')']`". Sequential representation maintains the context of the source code, which is beneficial for capturing the syntactic structure of the source code.

Structural representation considers the source code as a graph. It captures the relationship of different components (e.g., parameters, operation names, and types) in a code while preserving its syntax rule. The graph [80, 108] can be a *abstract syntax tree (AST)*, a *control flow graph (CFG)*, and a *data flow graph (DFG)*. We can better learn the source code's semantic information by using structural representation. Indeed, *graph neural networks (GNNs)* is recently popular in source code analysis and is based on the structural representation of source code. Recently, structural representation has enabled the use of GNNs for source code analysis.

2.1.3 Data Augmentation

Despite *Deep Neural Networks (DNNs)* having shown significant advantages, they meet two main challenges that hinder DNN from achieving high performance: 1) insufficient high-quality labeled training data, 2) discrepancies in data distribution between training and testing Data. A general solution to address these two issues is to augment the training data by increasing its size and diversity. Data augmentation [74] is a technique that aims to generate additional synthetic training data by automatically modifying existing data without requiring additional manual effort. Generally, data augmentation includes a series of well-designed data transformation methods. For instance, in *computer vision (CV)*, commonly used data augmentation methods include re-scaling, zooming, random rotating, padding, and adding noise [68].

Inspired by its great success in other domains, especially CV and NLP, software engineering researchers begin to apply data augmentation to solve machine learning-based source code tasks [2, 8, 20, 61, 79, 96], and the proposed methods are known as *code refactoring* shown in Figure 2.3. Code refactoring is a set of techniques that involves restructuring source code syntax while preserving its underlying semantic information. These techniques were originally used to simplify code and improve its readability and maintainability [40, 45]. General 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 rewrites the syntax rules of the source code but does not change the semantic behavior of the program in a way that changes it. However, existing studies [5, 97] have shown that simple strategies such as code refactoring have limited advantages in improving the code model perfor-

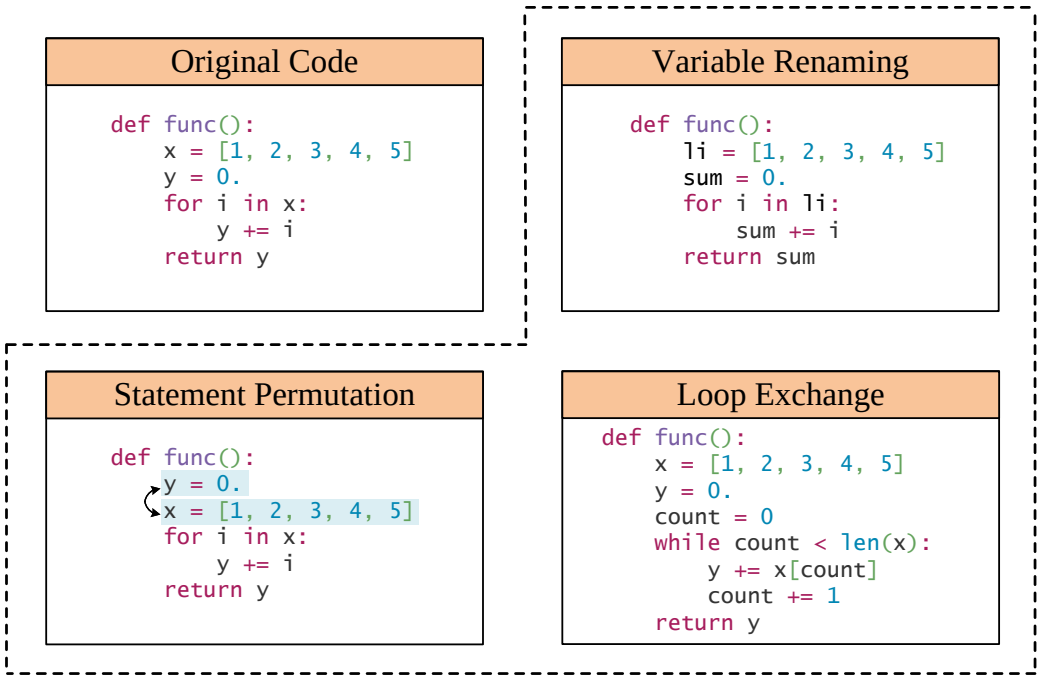


Figure 2.3: An example of data augmentation to transform code snippets, Wang *et al.* [81])

mance. To overcome these limitations, inspired by the analogy of source code to texts and graphs (as mentioned in Section 2.1.2), in our initial investigation, we aim to determine whether data augmentation methods originating from NLP, designed for text data, and graph learning, designed for graph data, can effectively increase the variety of training data for source code learning.

2.1.4 Robustness Testing of Source Code Models

Robustness is a critical property of *deep learning (DL)* models that assess how well the model can handle variations in the distribution of training data. Typically, there are two categories of robustness for DL models: natural robustness and adversarial robustness. Natural robustness examines how models perform when exposed to natural distribution shifts, such as changes in the data environment and newly collected data. Adversarial robustness focuses on a model's ability to withstand adversarial attacks.

In recent studies, researchers have introduced distribution-shifted code data, which consists of code with the same functionality but written by different programmers, to

investigate the natural robustness of code models. Additionally, various adversarial attack methods have been developed to evaluate the adversarial robustness of code models. For instance, Yang *et al.* [95] proposed a method that alters variable names in the code to force the model into making incorrect predictions. In our work, we mainly assess the adversarial robustness of our trained models using adversarial attack methods specifically designed for code. We do not explore natural robustness, as the existing datasets for natural robustness do not fully support the code-related tasks we investigate, including authorship attribution, bug detection, and code clone detection.

2.2 Related Work

Data augmentation has made remarkable success in the field of machine learning. Indeed, data augmentation has proven to be a valuable technique for improving model performance, specifically increasing trained model accuracy and enhancing dataset diversity [63]. Inspired by that code can be represented as text data [1], we review related works from four perspectives, source code learning enhancement, empirical study for source code learning, data augmentation for source code learning, and data augmentation for natural language processing, graph learning, and computer vision.

2.2.1 Source Code Learning Enhancement

Our goal in this research is to enhance the performance of models related to dealing with source code tasks. Previous studies with similar aims have approached this problem using various methods and techniques.

Self-supervised learning. Despite source code models having made significant progress in software engineering research, there are still some limitations. One such limitation is that some code models are designed to solve specific problems, making it challenging to extend their code representations to address other issues. To enhance the flexibility of these code models, Bui *et al.* [7] introduced a self-supervised learning mechanism for solving the code representation. This approach utilizes the prediction of identified sub-trees as the self-supervised task for training code representations. Consequently, the resulting code representations can be applied to various downstream tasks.

Pre-trained models fine-tuning. Reusing *pre-trained models* to address code analysis problems is a straightforward approach that has gained significant attention. Several

source code models have been developed for this purpose. For instance, Kanade *et al.* [39]. introduced *CuBERT*, a model architecture similar to *BERT*, which was trained on Python source code. Buratti *et al.* [9] employed a transformer-based model trained on C language to create the pre-trained model *C-BERT*. These models, *CuBERT* and *C-BERT*, were trained using single programming languages. More recently, there has been a move towards multi-programming language pre-trained models. Lu *et al.* [51] introduced *CodeGPT*, trained in Python and Java. Feng *et al.* [19] presented *CodeBERT*, a pre-trained model capable of learning from six programming languages and natural language text. For capturing the semantic structure of code, Guo *et al.* [25] proposed GraphCodeBERT, which incorporates data flow information from code during the pre-training stage.

In contrast to these previous efforts, our work primarily concentrates on enhancing the model training process through data augmentation. Consequently, the data augmentation methods proposed in our works can be applied to any code classification task independent of pre-trained models.

2.2.2 Empirical Study on Source Code Learning

Recent years have seen numerous empirical studies within the domain of *ML4Code*. Chirkova *et al.* [11] conducted an extensive empirical study evaluating the use of *Transformers* for source code learning tasks, including code completion, program repair, and bug fixing. Siow *et al.* [69] conducted a study to assess existing program representation techniques. In contrast, Zhang *et al.* [103] analyzed testing and debugging practices for machine learning programs, emphasizing the potential for platform execution issues to cause failures. Jebnoun *et al.* [37] performed a comparative study examining the distribution of code smells between machine learning and traditional applications. Yan *et al.* [94] conducted a comprehensive empirical study on code search using machine learning techniques, showing their effectiveness in code reuse queries. More recently, Hu *et al.* [34] delved into the distribution shift problem in code learning, identifying five types of shift in code data. Steenhoek *et al.* [71] reproduced nine DNN models and two vulnerability detection datasets, contributing to the understanding of deep learning-based models in source code learning. Mastropaolo *et al.* [55] conducted a comprehensive empirical study evaluating the robustness of Github Copilot's code completion approach. Niu *et al.* [58] performed a comparative study analyzing newly

developed pre-trained PL models to advance understanding in source code learning.

Unlike these existing empirical studies, our work mainly focuses on data augmentation in source code learning, which has received relatively little attention thus far.

2.2.3 Data Augmentation for Source Code Learning

Data augmentation has been a highly successful technique in machine learning, and its application has extended to big code tasks to enhance the performance of code models in terms of accuracy and robustness [17, 68]. Adversarial training [23], which involves introducing adversarial examples to the training data, has been explored as a data augmentation method in code learning. Zhang *et al.* [104] introduced a code data augmentation approach utilizing the *Metropolis-Hastings modifier (MHM)* algorithm [100] to improve deep comment generation models. Mi *et al.* [57] generated additional data using *Auxiliary Classifier Generative Adversarial Networks (GANs)*. Additionally, code refactoring, a program transformation method designed specifically for code, has been adopted as a mainstream data augmentation method for code. Yu *et al.* [97] devised program transformation rules for Java and evaluated their effectiveness in three major code-related tasks. Allamanis *et al.* [2] employed four simple code rewrite rules as data augmentation techniques to enhance code model generalization.

In contrast to the aforementioned works, our study is unique as it is the first to comprehensively assess the effectiveness of various types of data augmentation methods for code learning, methods that are tailored to code data, text data, and graph data.

2.2.4 Data Augmentation for Natural Language Processing

Data augmentation for NLP is typically categorized into three groups: *paraphrasing*, *noising-based methods*, and *sampling-based methods* [17]. An example of paraphrasing is the back-translation technique introduced by Xie *et al.* [92], which involves translating the original text into another language and then back to the original language. Paraphrasing is popular in NLP as it conveys the same information as the original text but in a different form. Noising-based methods, such as *easy data augmentation (EDA) et al.* [85], introduce subtle noise into the original data while maintaining its semantic meaning. For instance, *Random Swap*, a method from EDA, randomly selects two words in a sentence and swaps their positions. Sampling-based methods differ from paraphrasing and noising-based techniques and are task-specific, requiring

both data and label formats. Inspired by *Mixup*, Guo *et al.* [26] adapted this approach from image data to text data. They generated synthetic data by linearly mixing the embeddings rather than directly manipulating the raw text data. To address the limitations of traditional methods, like "Synonym Replacement," which often require substantial prior knowledge and can produce suboptimal results, Ren *et al.* [65] introduced *Text AutoAugment*. This method treats a combination of various text data augmentation operations as an augmentation policy and uses an efficient Bayesian Optimization algorithm to discover the best policy automatically.

Our work is distinct from data augmentation methods used in NLP because it concentrates on guiding code model training. This work encompasses existing data augmentation techniques originating from code data, data augmentation methods adapted from text data, and even newly developed program transformation methods. It provides a flexible and evolving framework for enhancing code model training.

Chapter 3

Boosting Source Code Learning with Text-Oriented Data Augmentation

3.1 Introduction

In recent years, we have witnessed a significant upsurge in the application of *machine learning to big code (ML4Code)* [1]. This trend is primarily driven by the formidable capabilities of machine learning, particularly deep learning, in uncovering intricate patterns within extensive code data. The promising outcomes of ML4Code in various downstream code-related tasks, including but not limited to code clone detection[1], vulnerability detection [50], and problem classification [62], underscore its immense potential in practice. To develop high-performance programming language (PL) models, two critical elements are indispensable: high-quality training data and meticulously crafted model architectures. While insights from the natural language processing (NLP) domain can be instrumental in shaping model architectures, it's crucial to underscore that the quality of the training data plays an equally pivotal role in the model's performance [1, 33]. Preparing high-quality training data is an arduous process that cannot be circumvented. Typically, this involves data collection, thorough data cleaning, and meticulous manual labeling before the data can be effectively utilized. These steps are inherently challenging, with data labeling being particularly labor-intensive and time-consuming. To illustrate the extent of this challenge, consider that even labeling data from just four code libraries can consume as much as 600 person-hours [108]. Consequently, acquiring an ample volume of high-quality training

data for programming language (PL) models remains a persistently challenging issue.

In fields like *computer vision (CV)*, data augmentation has proven to be a widely adopted strategy to address the challenge of limited labeled training data. The fundamental concept behind data augmentation is to generate new training data by altering existing labeled data while ensuring that the modified data retains the original semantics. For instance, when training a model for image classification, instead of relying solely on the original training dataset, it is common practice to employ image transformation techniques. These techniques create a more diverse set of images for training the model [29]. Existing studies [60, 63] have demonstrated that data augmentation can be highly effective in enhancing the trained models' accuracy and robustness.

Although data augmentation has received significant attention and shown its worth in other fields, its potential and effectiveness in the context of source code learning remain to be fully understood. This raises the question of how data augmentation can impact the performance of models designed for source code-related tasks. In the realm of ML4Code, the majority of studies have primarily focused on enhancing model architectures [25], refining code representation techniques [3, 89], and innovating learning strategies to improve the performance of code-related tasks. While these endeavors have certainly advanced the field, relatively few works have explored the problem of automatically enriching training data for these tasks. Indeed, there are a few instances, such as the works such as [2, 97], where data augmentation through code refactoring methods was proposed. However, these methods have been found to have a limited impact on performance, highlighting the need for more effective data augmentation approaches in the domain of source code learning.

Our Contributions. In the context of source code learning, code is often represented as sequential data when used as training data. Our study draws inspiration from these sequential representations. It delves into an empirical investigation to determine whether existing data augmentation techniques borrowed from the field of NLP, which typically handles text data, can effectively enhance the quality of training data in source code learning. Concretely, we begin by surveying existing data augmentation methods in the literature. Subsequently, we categorize these methods to understand their applicability to source code learning. As a result, we identify seven data augmentation methods used initially in NLP but appear to have potential applications in source code learning. These methods are then adapted to train code models. We then evaluate the performance of these adapted data augmentation methods. Specifically, we gauge their

impact on improving the accuracy and robustness of code models. We open source all implementations as well as data and models[3] to support future research on data augmentation for source code learning.

We designed the large-scale empirical study to answer the following three research questions:

- RQ1: Can existing data augmentation methods produce accurate code models?
- RQ2: Can existing data augmentation methods produce robust code models?
- RQ3: How does data volume affect the effectiveness of data augmentation methods?

3.2 Methodology

3.2.1 Overview

As introduced in Section 2.1.2, source code can be represented as sequential or structural data. Therefore, inspired by these comparable relationships, we assess the effectiveness of the data augmentation methods from NLP in source code learning. Specifically, 7 data augmentation methods are collected from NLP as shown in Fig. 3.1.

We collected 18 program transformation methods from the existing works, including 10 code refactoring operators, e.g., *local variable renaming* and *argument adding* [61] and 8 code transformation methods, e.g., *duplication* and *unreachable loops/branches* [86]. Comprehensive information about these program transformation methods and their implementations is available on our project website. During our experiments, we randomly selected one of these 18 program transformation methods for each code data set and applied it to the original code, thus generating augmented code data.

3.2.2 Adapting Data Augmentation Methods for Source Code Learning

In this section, we will explain the adaptations made to these 7 data augmentation methods to suit source code data. This is essential for understanding how these meth-

[3] https://sites.google.com/view/dataaug4code

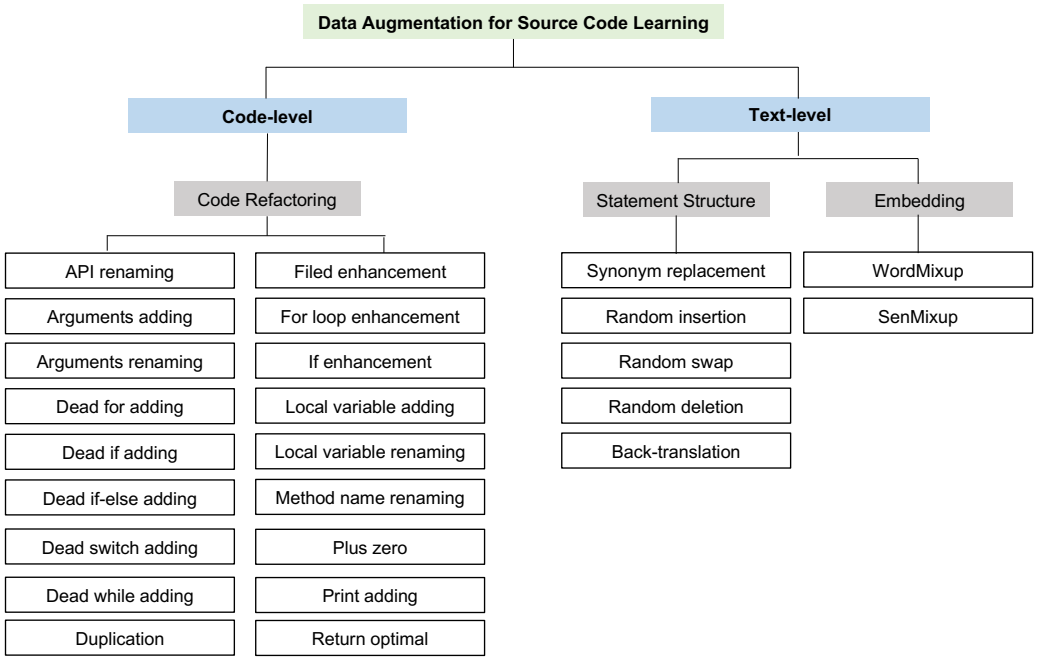
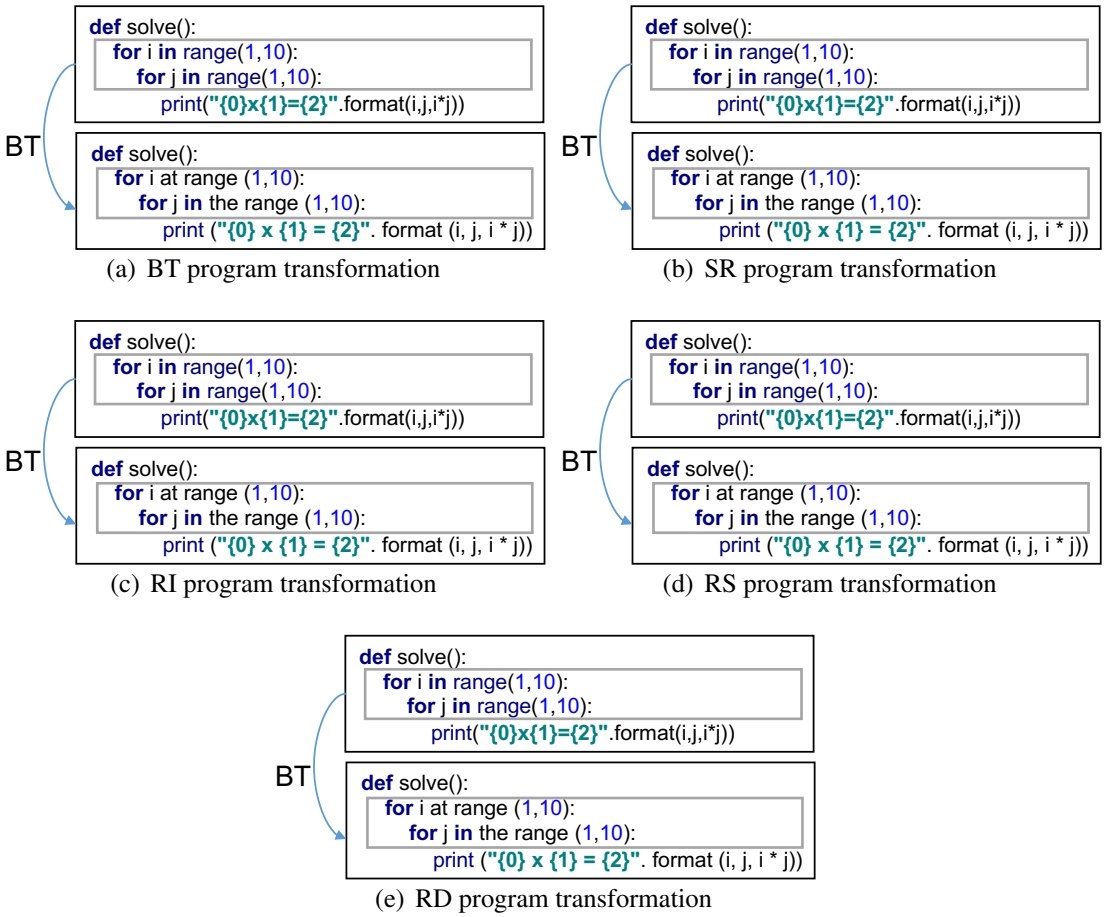


Figure 3.1: Data augmentation methods for source code learning

ods were applied in our study. We will also offer specific details on the modifications made to each method to make them compatible with source code data.

Back-translation (BT). This method translates the original source code into another language and then back to the original language, thus generating additional source code data. A program P with k lines is represented as $P = \{S_1, S_2, \ldots, S_k\}$, where S represents the tokenization of the statement and S_k specifically is a statement S at the kth line in a program. In our experiment, BT is used as one program transformation method to produce the transformed code data $P_{bt} = \left\{S_1^{bt}, S_2^{bt}, \ldots, S_k^{bt}\right\}$ from its original code data P. Specifically, we apply the English-French translation model (in both directions) [92] to perform back-translation on each statement $S_i\,(S_i \in P)$ and obtain its paraphrases $S_i^{bt}\left(S_i^{bt} \in P_{bt}\right)$, where S_i^{bt} is the new transformed statement generated by the bi-directional neural machine translation model. For example, in Fig. 3.2(a), after the BT operator, we change the statement "`in range`" to "`at range`" and "`in the range`" respectively. Note that while these alterations might slightly disrupt the syntax of the code data, they are minor, and the fundamental relationship between the features and the label remains intact.

Synonym Replacement (SR). In source code analysis, we randomly select n state-



(a) BT program transformation

(b) SR program transformation

(c) RI program transformation

(d) RS program transformation

(e) RD program transformation

Figure 3.2: 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.)

ments from a program. Next, each of the n words is replaced with a synonym, randomly chosen. As Fig. 3.2(b) shows, we randomly select one statement from a program and replace it with another string generated from its randomly chosen synonyms. Like the BT method we introduced, this approach maintains the original relationship between the features and the code's label.

Random Insertion (RI). We begin by selecting a random synonym for a word in the chosen statement. This synonym is then randomly inserted into a chosen position in the statement. This process is typically repeated n times. As Fig. 3.2(c) presents, we

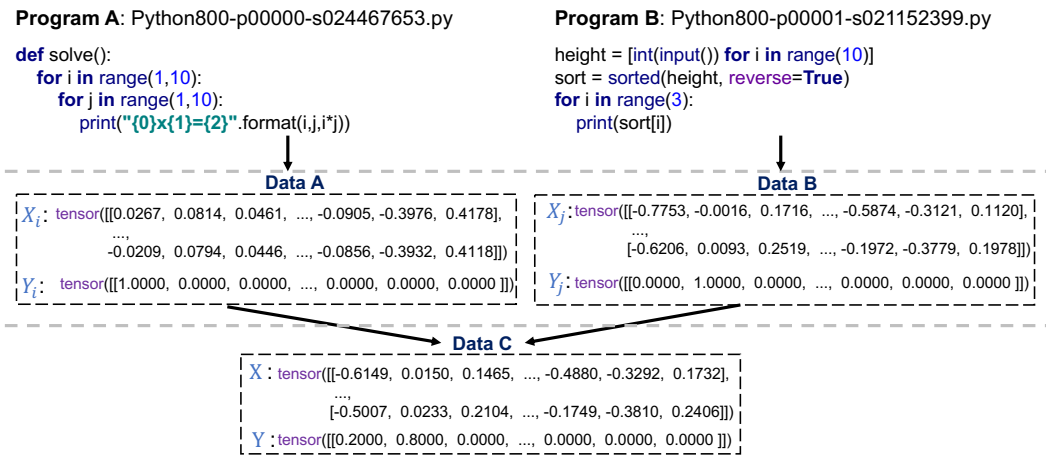


Figure 3.3: An example of linear interpolation of two programs

randomly insert a synonym-generated string at a random position within the selected statement from the original code, e.g., "`for i in range(1,10)`". This method effectively maintains the original relationship between the code's features and labels.

Random Swap (RS). In ML4Code, we randomly select two statements within a program and swap their positions. This process is typically repeated n times. As Fig. 3.2(d) shows, we randomly select two different statements "`for j in range(1,10)`" and "`print("{0}x{1}={2}".format(i,j,i*j))`" and exchange their positions. Although we randomly select two different statements in a program and swap their positions, this program transformation is also conducted under a label-preserving situation.

Random Deletion (RD). We randomly remove words from a randomly selected statement. Given an example in Fig. 3.2(e), we delete words in a statement with a probability of $p = 0.01$. For instance, after the *RD* program transformation, the operator "`in`" is removed from the statement "`for i in range(1,10);`". This method also preserves the original relationship between the features and the label in the code.

In the following, we will present data augmentation methods that are modified from *Mixup* [99], a widely-used data augmentation method in CV. Specifically, *Mixup* generates new image data by linearly mixing the features and labels of two selected images, which has inspired the development of various data augmentation methods across different domains. Mixup's principles have been adapted into NLP as WordMixup and SenMixup, which we introduce below.

WordMixup and SenMixup. Originally, *WordMixup* operates by interpolating sam-

ples within the word embedding space, while *SenMixup* performs interpolation within the hidden states of sentence representations, as described in previous work [26]. We've made slight modifications to apply WordMixup and SenMixup to source code learning. In our study, as Eq. (3.1) shows, we employ two variants of Mixup, each with distinct characteristics. The first variant, labeled as WordMixup, performs interpolation within the embedding space of statement representations. The second variant, named SenMixup, conducts interpolation after a linear transformation just before the data is passed through 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 is the code data, and y^i and y^j are their labels, the generated new data are obtained via *WordMixup* and *SenMixup*, as follows:

$$\begin{aligned} x^{ij}_{WordMix} &= \lambda x^i + (1-\lambda)x^j \qquad & x^{ij}_{SenMix} &= \lambda f(x^i) + (1-\lambda) f\left(x^j\right) \\ y^{ij}_{WordMix} &= \lambda y^i + (1-\lambda)\, y^j \qquad & y^{ij}_{SenMix} &= \lambda y^i + (1-\lambda)\, y^j \end{aligned} \tag{3.1}$$

SenMixup follows a similar operation with [26], and $f(\cdot)$ represents a linear transformation operator that male input data and the output data have the same dimension. Moreover, $x^{ij}_{WordMix}$ and x^{ij}_{SenMix} donate the new generated training data obtained by *WordMixup* and *SenMixup* respectively, and $y^{ij}_{WordMix}$ and y^{ij}_{SenMix} are labels. The parameter λ are the *Mixup* ratio, and according to [99], it is sampled from a *Beta* distribution with a shape parameter $\alpha\left(\lambda \sim Beta\left(\alpha, \alpha\right)\right)$.

3.3 Evaluation

All implementations are done in Python, ensuring the extensibility of this study for future techniques. Additionally, we offer a code refactoring generator that includes 18 different code refactoring methods compatible with both Java and Python. Our models employ TensorFlow 2.3 and Keras 2.4.3 for BagOfToken and SeqOfToken. PyTorch 1.6.0 is used for CodeBERT and GraphCodeBERT. We set the training epoch to 50 for these four models. We use a default Mixup ratio of 0.1 to augment training data with Mixup. To reduce the impact of randomness, we train each model five times and report the average results, along with the standard deviation. All experiments were conducted on a server with 4 NVIDIA RTX A6000 GPUs.

Table 3.1: 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	

Code classification serves as a fundamental process for automatically assessing the functionality of programs, a vital aspect of software reuse [62]. Our initial emphasis is on code classification. In Table 3.1, you can find details about the datasets and models used. To ensure consistency, we adhere to the data partition settings provided by the official projects, which involve dividing the data into training, validation, and test sets.

Problem classification is a standard task in source code learning that involves classifying the intended functions of source code. In this task, the model must identify the specific problem that a given piece of code aims to address. We utilize two recently introduced datasets, Java250 and Python800 [62]. The Java250 dataset comprises 250 problems, while the Python800 dataset extends to 800 problems.

Bug detection is primarily concerned with identifying defects or issues within a codebase, typically framed as a binary classification problem. Our study employs two real-world datasets, Refactory [36] and CodRep1 [107], designed for bug repair tasks.

Authorship attribution pertains to identifying the author of a specific code fragment by discerning programmer-specific characteristics in their published source code. This process is essential for appropriately recognizing a programmer's contributions and can also be instrumental in plagiarism detection. Our dataset, sourced from GCJ and made available by the existing work [95], is utilized for this purpose.

Clone detection is a critical task that seeks to determine the semantic similarity between two code fragments, preventing bug propagation and simplifying software maintenance. For this purpose, we employ the clone detection benchmark dataset BigCloneBench [73]. It's important to note that Mixup and its variants do not apply to this task, as their input is in the form of code pairs.

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

Study Design. Accuracy is a basic metric that calculates the % of correctly classified data over the entire test data [17, 47]. Here, clean accuracy (*Accuracy* in Eq.(3.2)) means the accuracy of models on the original test data.

$$Accuracy = \frac{\#Corrected\,classified\,data}{\#Entire\,data}\% \tag{3.2}$$

Therefore, we begin by examining whether data augmentation methods derived from NLP can enhance the accuracy of code models. These methods are compared against training scenarios without data augmentation and training using code refactoring methods (as detailed in Section 3.2.1). In this research question, we initially prepare the original training data and initialize code models with random parameters. Subsequently, we train these models using various data augmentation methods, as outlined in Fig. 3.1. We assess these methods based on two key metrics: 1) the speed of model convergence and 2) the final accuracy achieved by the model within fixed training epochs.

Results Analysis. As demonstrated in Table 3.2, *SenMixup* consistently achieves the highest performance in 6 out of 12 cases. Interestingly, it is surprising that in most instances (8 out of 16), code refactoring methods fail to enhance model accuracy compared to *No Aug*. For SeqofToken models, once again, *SenMixup* outperforms *No Aug*, with accuracy improvements of up to 8.74% and an average improvement of 4.63%, as well as *Refactor*, with accuracy improvements of up to 4.53% and an average improvement of 2.66%. These results suggest that *SenMixup* is a viable choice for conventional deep neural network models (e.g., FNN, CNN) when addressing code classification tasks.

Moving our focus to pre-trained PL models, we initially noticed that *SenMixup* doesn't significantly enhance the accuracy of pre-trained models, with only a maximum improvement of 1.18%. In contrast, data augmentation methods that introduce slight syntax alterations to source code, such as *RS*, prove more effective, resulting in clear accuracy improvements, up to 2.25% when compared to *No Aug*, and up to 3.01% when compared to *Refactor*. In the case of GraphCodeBERT, *RS* delivers the highest accuracy improvements in three out of four datasets. However, despite *SenMixup* and

Table 3.2: Accuracy ↑ (average ± standard deviation, %) of BagofToken, SeqofToken, CodeBERT and GraphCodeBERT. **No Aug (Baseline)**: without data augmentation. The best results are highlighted in gray.

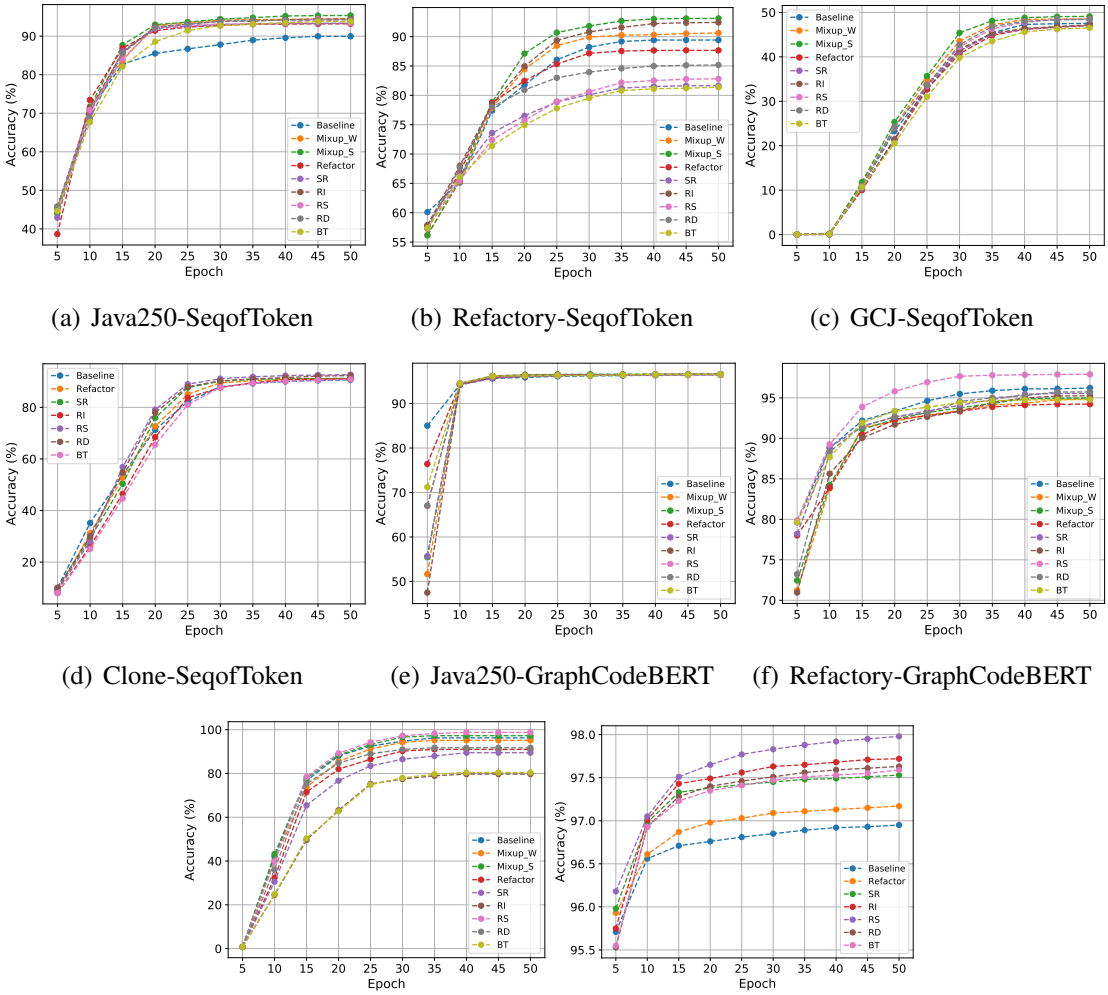
Model	DA method	Java250	Refactory	GCJ	BigCloneBench	Model	DA method	Java250	Refactory	GCJ	BigCloneBench
	No Aug	71.24 ± 0.04	85.15 ± 0.12	27.82 ± 0.14	85.23± 0.34		No Aug	96.39 ± 0.03	96.22 ± 0.11	90.98 ± 0.14	96.89 ± 0.19
	WordMixup	73.12 ± 0.01	86.46 ± 0.28	28.43 ± 0.25	-		WordMixup	96.31 ± 0.04	96.16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

RS yielding relatively better results than other methods, our statistical analysis involving the *T-Test* and *Wilcoxon signed-rank test* (for detailed results, refer to our project site) indicates that their advantage is not statistically significant. Specifically, in only two out of 28 cases, the p-values are less than 0.05. This finding underscores the potential for the development of more potent data augmentation methods for code learning

On the other hand, we also examined the convergence speed of models employing various data augmentation methods. Fig. 3.4 visually illustrates the outcomes derived from the training logs. These results align with our earlier findings concerning the final accuracy of models. Two methods, namely *SenMixup* and *RS*, outperform the others in terms of convergence speed. This suggests that, even with constraints on computational resources, these two methods are practical recommendations.

3.3.2 RQ2: Can existing data augmentation methods produce robust code models?

Study Design. Robustness is another crucial metric for assessing the generalization capability of a trained model, particularly its ability to handle previously unseen data [5]. In this RQ, we delve into the adversarial robustness of multiple trained code models. Drawing from literature [63], which indicates that data augmentation enhances the adversarial robustness of DNN models in various domains (such as CV), we aim to ascertain whether this conclusion extends to code models. To achieve this, we adopt two contemporary adversarial attack techniques: the naturalness-aware attack



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

(d) Clone-SeqofToken (e) Java250-GraphCodeBERT (f) Refactory-GraphCodeBERT

(g) GCJ-GraphCodeBERT (h) Clone-GraphCodeBERT

Figure 3.4: Training log of models (SeqofToken and GraphCodeBERT) in Java250, Refactory, GCJ, and BigCloneBench.

(ALERT)[95] and the Metropolis-Hastings modifier (MHM) algorithm[100]. These attacks allow us to evaluate the improvement in robustness of two pre-trained programming language (PL) models that were trained using data augmentation.

$$ASR = \frac{\#Successfully\,generated\,adversarial\,data}{\#Corrected\,classified\,test\,data}\% \tag{3.3}$$

Results Analysis. Adversarial robustness is a critical metric that assesses a model's ability to handle data with noise, an important characteristic that should be evaluated.

Table 3.3: ASR ↓ (%) on test data. **No Aug (Baseline)**: without data augmentation. The best results are highlighted in gray. The victim DNN models are CodeBERT and GraphCodeBERT.

Model	DA method	Java250		Refactory		GCJ		BigCloneBench	
		MHM	ALERT	MHM	ALERT	MHM	ALERT	MHM	ALERT
CodeBERT	No Aug	40.85	53.46	35.21	42.51	33.33	56.11	9.67	24.52
	WordMixup	40.68	53.85	37.54	48.98	32.15	53.34	-	-
	SenMixup	38.13	51.19	32.24	41.13	29.27	52.56	-	-
	Refactor	38.34	52.23	38.89	48.78	35.54	52.89	8.58	18.34
	SR	41.64	54.51	39.78	53.85	35.48	52.69	12.49	26.03
	RI	43.45	55.87	38.64	45.34	40.79	65.79	8.29	30.02
	RS	39.21	51.09	39.02	46.78	28.23	52.42	13.53	30.27
	RD	38.79	52.11	29.54	40.91	35.64	55.45	9.51	30.11
	BT	41.98	53.19	63.63	69.69	40.37	55.96	11.34	26.36
GraphCodeBERT	No Aug	23.14	42.53	23.12	30.33	31.21	56.01	4.11	12.11
	WordMixup	23.12	42.44	22.19	28.32	31.67	56.45	-	-
	SenMixup	22.52	42.02	20.34	27.33	30.21	54.46	-	-
	Refactor	23.04	42.36	24.56	32.44	30.33	56.56	6.02	6.13
	SR	23.12	42.39	28.77	47.66	40.01	66.67	6.15	10.21
	RI	22.56	42.41	23.24	33.12	43.52	65.74	2.18	6.21
	RS	23.01	41.87	21.45	29.65	30.16	54.76	6.08	12.05
	RD	22.78	41.98	20.01	27.31	31.71	60.98	6.16	8.25
	BT	22.89	42.42	35.99	46.33	41.67	62.96	8.21	14.21

In Table 3.3, we present the *Attack Success Rate (ASR, in Eq.(3.3))* of two state-of-the-art attack methods on our trained models. These results shed light on the fact that data augmentation does not consistently enhance the robustness of models when compared to the *No Aug* models. This phenomenon aligns with the conclusion from previous work [96], indicating that simply increasing the training data is insufficient for improving the robustness of code models. Furthermore, it's crucial to note that, in some instances, while data augmentation can contribute to the training of a more robust model, the improvements in robustness are negligible. For example, the most significant improvement is merely 5.98% (observed in GraphCodeBERT-Refactor-BigCloneBench-ALERT).

Next, we proceed to compare each data augmentation method. In the case of CodeBERT, *RS* stands out as the top-performing method, showcasing relatively superior robustness improvement in three out of eight cases. It reduces ASR by up to 5.10% under the MHM attack and 3.69% under the ALERT attack when compared to the *No Aug* condition.

Table 3.4: Accuracy ↑ (average ± standard deviation, %) of CodeBERT using #% training data. **No Aug (Baseline)**: without data augmentation. The best results are highlighted in gray.

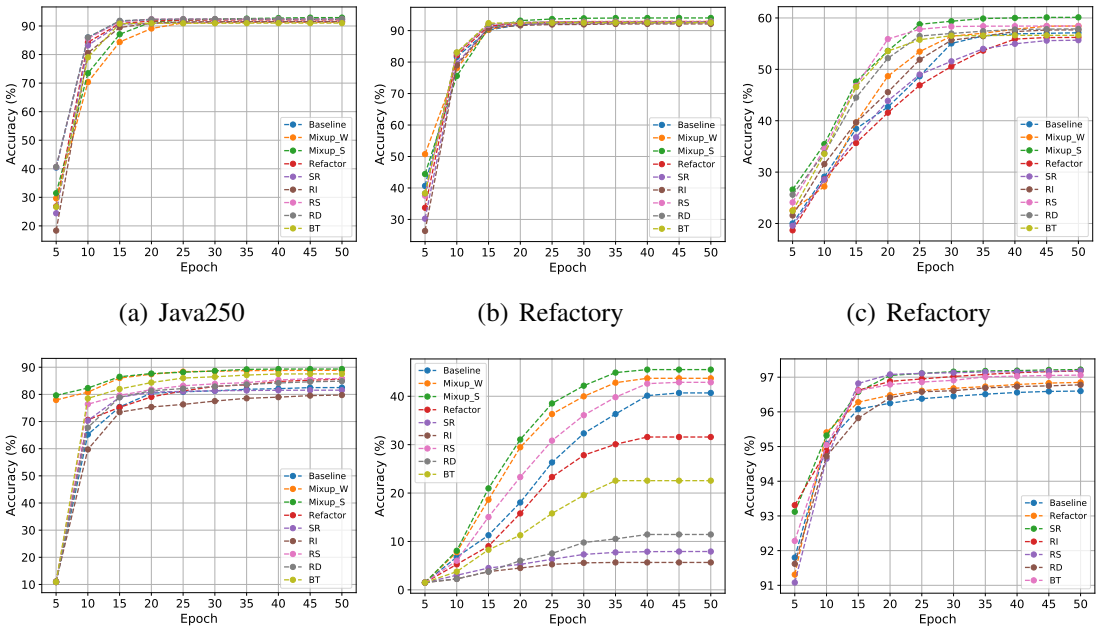
Model	DA method	Java250	Refactory	BigCloneBench	Model	DA method	Java250	Refactory	BigCloneBench
CodeBERT (10%)	No Aug	92.28 ± 0.03	82.51 ± 0.18	96.58 ± 0.15	CodeBERT (3%)	No Aug	78.15 ± 0.13	75.89 ± 0.23	78.67 ± 0.44
	WordMix	91.33 ± 0.04	83.89 ± 0.13	-		WordMix	78.89 ± 0.25	87.33 ± 0.32	-
	SenMixup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

In the context of GraphCodeBERT, it's noteworthy that *RD* proves to be the most effective choice for enhancing robustness in two out of eight cases. This method reduces ASR by up to 3.11% in the MHM attack and 3.86% in the ALERT attack when compared to the *No Aug* scenario. Intriguingly, concerning pre-trained PL models, it appears that methods such as *RS*, which may slightly disrupt the syntax of programs, can yield more accurate and robust code models for solving downstream tasks compared to *Refactor*. This observation could inspire future research, suggesting that there's no strict need to adhere to program constraints when designing new data augmentation methods. Based on these findings, we recommend utilizing *RS* for augmenting the training data when applying pre-trained PL models to source code learning. This method exhibits the most significant improvements in robustness across three out of four tasks.

3.3.3 RQ3: How does data volume affect the effectiveness of data augmentation methods?

Study Design. To assess the effectiveness of data augmentation methods in addressing the issue of limited labeled data, we've reduced the size of the training set and repeated the evaluations, as performed in RQ1 and RQ2.

Results Analysis. Data augmentation methods play a crucial role in expanding the



(a) Java250 (b) Refactory (c) Refactory

(d) GCJ (e) GCJ (f) BigCloneBench

Figure 3.5: Training log of CodeBERT using 10% training data.

size and enhancing the diversity of training data, particularly in situations with limited labeled training data. Therefore, it's essential to investigate whether data augmentation methods improve the accuracy and robustness of DNN models when the available training data is limited in size. In this research question, we deliberately reduce the training data size to 10%, 5%, 3%, and 1% of the original dataset. For the authorship attribution task, we retain 10% and 50% of the original training data due to its relatively small size. We then reiterate the experiments conducted in previous Sections. Our focus in this particular section is on the results obtained from the CodeBERT models, while additional outcomes are accessible on our project website.

Table 3.4 and the left section of Table 3.6 provide an overview of the clean accuracy results. In the context of problem classification tasks like Java250, we observe that data augmentation with CodeBERT, when using only 1% of the training data, significantly improves accuracy in only one instance, showcasing a remarkable accuracy improvement of up to 5.64%. On the other hand, clone detection tasks don't see significant improvements in accuracy with data augmentation, as the gains are relatively small. However, the bug detection task demonstrates consistent accuracy improvements using the *SenMixup* method, showing significant improvements ranging from

3.32% to 12.92%. It's worth noting that two noising-based methods, *SR* and *RI*, which performed well with the entire training dataset, falter after the training data is substantially reduced. For example, in the Java250-CodeBERT (1%), the models generated by *SR* and *RI* exhibit accuracy rates of only 37.06% and 27.03%, while the baseline method (*No Aug*) achieves an accuracy of 49.62%. To summarize, *SenMixup* remains the most recommended method, although its advantages are not statistically significant according to our analysis (p-values are more significant than 0.05). This highlights the continued utility of *SenMixup* even in scenarios with severely limited training data.

The training logs in Figure 3.5 demonstrate that the convergence speed of models is notably impacted when data augmentation methods are employed with reduced training data. For instance, in the case of GCJ-CodeBERT after ten epochs, data augmentation methods, including *RS*, *SenMixup*, and *WordMixup*, lead to more substantial accuracy improvements compared to the results of model training with the complete dataset. Surprisingly, for Refactory-CodeBERT, the model training using *SenMixup* and *WordMixup* brings almost 70.00% significant accuracy improvement at 5^{th} epoch compared to other data augmentation methods as well as *No Aug*. Compared to other data augmentation methods, models training using linear interpolation methods including *SenMixup* and *WordMixup* require more epochs to reach convergence, most data augmentation methods bring limited accuracy improvement after 30 epochs (except *SenMixup*). This phenomenon is similar to the findings in [98, 99], which reveals that data augmentation methods that strongly increase the complexity of training data require more epochs to converge.

Tables 3.5 and Table 3.6 present the results of ASR for MHM (Metropolis-Hastings Modifier) and ALERT (Naturalness-Aware Attack) attacks, respectively. In the case of the program classification task (Java250), a noticeable trend emerges as the size of the dataset decreases from 10% to 1%. The robustness improvements achieved by data augmentation methods become more pronounced, with the most substantial reduction in ASR observed in the CodeBERT (10%) dataset, dropping from 0.73% to 6.86% under MHM attack. For the clone detection task, most data augmentation methods that introduce minor structural changes to the code (except for RS and RD) do not significantly enhance robustness when the dataset size shrinks (as in the CodeBERT (1%) dataset). A similar pattern emerges in the bug detection task (Refactory). In contrast, for the authorship attribution task (GCJ), data augmentation methods that slightly alter the code's syntactic structure, especially RI, effectively enhance robustness. For

Table 3.5: ASR ↓ (%) of CodeBERT using limited training data on test data. The training data volume drops to 10%, 5%, 3%, and 1% of the original, respectively. **No Aug (Baseline)**: without data augmentation. The best results are highlighted in gray.

Model	DA method	Java250		Refactory		BigCloneBench		Model	DA method	Java250		Refactory		BigCloneBench	
		MHM	ALERT	MHM	ALERT	MHM	ALERT			MHM	ALERT	MHM	ALERT	MHM	ALERT
CodeBERT (10%)	No Aug	25.69	39.79	36.37	54.54	15.56	26.33	CodeBERT (3%)	No Aug	45.01	62.78	56.32	68.56	22.45	48.41
	WordMix	25.12	39.81	32.56	47.63	-	-		WordMix	41.67	55.32	55.32	74.69	-	-
	SenMixup	26.89	37.96	28.87	39.43	-	-		SenMixup	40.32	49.67	53.55	67.44	-	-
	Refactor	25.11	38.57	39.13	52.17	15.02	26.14		Refactor	41.25	45.53	62.56	66.43	10.87	19.51
	SR	26.32	43.42	52.17	47.83	17.33	34.42		SR	46.45	68.72	62.87	75.44	13.64	23.68
	RI	28.76	44.98	65.38	73.08	20.52	28.35		RI	49.82	70.44	54.11	65.88	24.76	38.09
	RS	25.02	38.83	60.12	70.01	16.92	28.06		RS	45.88	61.45	55.32	66.21	10.63	19.05
	RD	24.96	38.78	38.09	42.86	14.87	24.26		RD	44.78	61.75	54.34	64.42	22.15	43.91
	BT	29.76	39.65	30.76	46.15	14.81	30.38		BT	51.38	72.87	54.88	72.67	23.87	35.62
CodeBERT (5%)	No Aug	34.74	53.59	47.83	56.52	17.27	28.76	CodeBERT (1%)	No Aug	80.23	82.34	70.44	83.65	10.41	22.78
	WordMix	34.32	53.42	39.62	58.11	-	-		WordMix	76.72	79.32	69.11	78.31	-	-
	SenMixup	33.52	53.15	33.78	50.53	-	-		SenMixup	75.26	77.32	68.32	76.33	-	-
	Refactor	34.45	53.35	48.83	58.82	19.67	28.57		Refactor	82.59	86.63	72.23	79.42	10.93	22.94
	SR	36.98	54.09	35.87	43.45	22.74	43.59		SR	82.36	86.55	78.33	94.42	9.81	38.42
	RI	39.46	57.87	55.33	59.67	24.42	53.66		RI	84.52	87.11	67.25	91.31	22.36	38.26
	RS	33.13	53.39	46.56	51.41	18.68	30.95		RS	75.45	82.42	69.56	78.43	10.14	36.67
	RD	32.39	53.34	42.83	53.51	16.46	27.91		RD	73.37	76.43	77.38	87.42	10.33	21.85
	BT	40.87	59.76	50.01	52.42	18.32	28.15		BT	82.22	86.23	65.32	77.42	21.47	35.89

Table 3.6: Accuracy ↑ (average ± standard deviation, %) and ASR ↓ (%) of CodeBERT using #% training data, respectively, on test data. **No Aug (Baseline)**: without data augmentation. Dataset: GCJ.

DA method	Test accuracy		Robustness-MHM		Robustness-ALERT	
	10%	50%	10%	50%	10%	50%
No Aug	40.69 ± 0.11	86.47 ± 0.15	58.49	28.07	69.91	52.63
WordMix	42.14 ± 2.01	84.21 ± 0.23	52.98	36.03	63.56	54.95
SenMixup	44.51 ± 1.31	86.98 ± 0.31	51.45	27.87	61.11	49.55
Refactor	31.58 ± 0.21	78.21 ± 0.17	58.54	33.01	75.61	60.19
SR	7.52 ± 0.15	57.89 ± 0.09	70.01	39.47	70.01	64.47
RI	3.76 ± 0.13	22.56 ± 0.15	20.02	37.93	60.01	48.28
RS	41.67 ± 0.17	86.98 ± 0.19	60.38	27.49	71.69	50.01
RD	12.78 ± 0.24	55.64 ± 0.14	41.18	43.84	41.18	60.27
BT	23.32 ± 0.27	69.17 ± 0.06	61.29	36.26	67.74	60.44

instance, RI reduces ASR by up to 38.47% in the CodeBERT (10%) dataset under MHM attack compared to the No Aug model. Intriguingly, in contrast to the results obtained with the entire training dataset, our statistical analysis reveals that, in most cases (14 out of 21), RI outperforms other methods significantly (with a p-value of less than 0.05). This finding suggests that RI is a suitable choice for building robust code models when training data is limited.

3.4 Discussion

3.4.1 Findings

The need for data augmentation when preparing code models hinges on several factors. Your trained models can benefit significantly if you employ a suitable data augmentation method, such as SenMixup. They exhibit higher accuracy (up to 12.92%) and robustness (up to 15.11%) compared to models without data augmentation. However, it's worth noting that when using pre-trained programming language models, the impact of data augmentation becomes less significant. This could be attributed to pre-training already serving as a form of data augmentation, enhancing the entire model training process. Consequently, additional data augmentation techniques might not be as beneficial as they are when pre-training isn't involved. An in-depth analysis of this phenomenon could be a promising avenue for future research.

Previous research has demonstrated that in the domain of natural language, text data can tolerate syntactic changes to some extent without losing readability and validity as additional training data. This is as long as these changes remain within a limited range that doesn't disrupt the original relationships between the text data and their labels. Noising-based data augmentation methods, such as RI and RS, have proven valuable in NLP [54]. This was highlighted in a recent study as well. In the realm of source code learning, our experimental results align with a similar conclusion. Even when data augmentation methods produce training data that slightly deviates from the syntax of the source code, it still proves valuable in enhancing the quality of source code learning. Our experiments reveal that pre-trained programming language models using the RS method can achieve higher accuracy (up to 14.94%) and greater robustness (up to 7.24%) compared to models without such augmentation.

3.4.2 What factors contribute to code models' improved accuracy and robustness when utilizing Mixup-based data augmentation methods?

As discussed in Section 3.4.1, Mixup-based data augmentation methods, such as SenMixup, can effectively improve the performance of source code models, especially in situations with limited datasets. Therefore, we further explore why mixup-based data

augmentation methods can exhibit superior performance in accuracy and robustness compared to other methods.

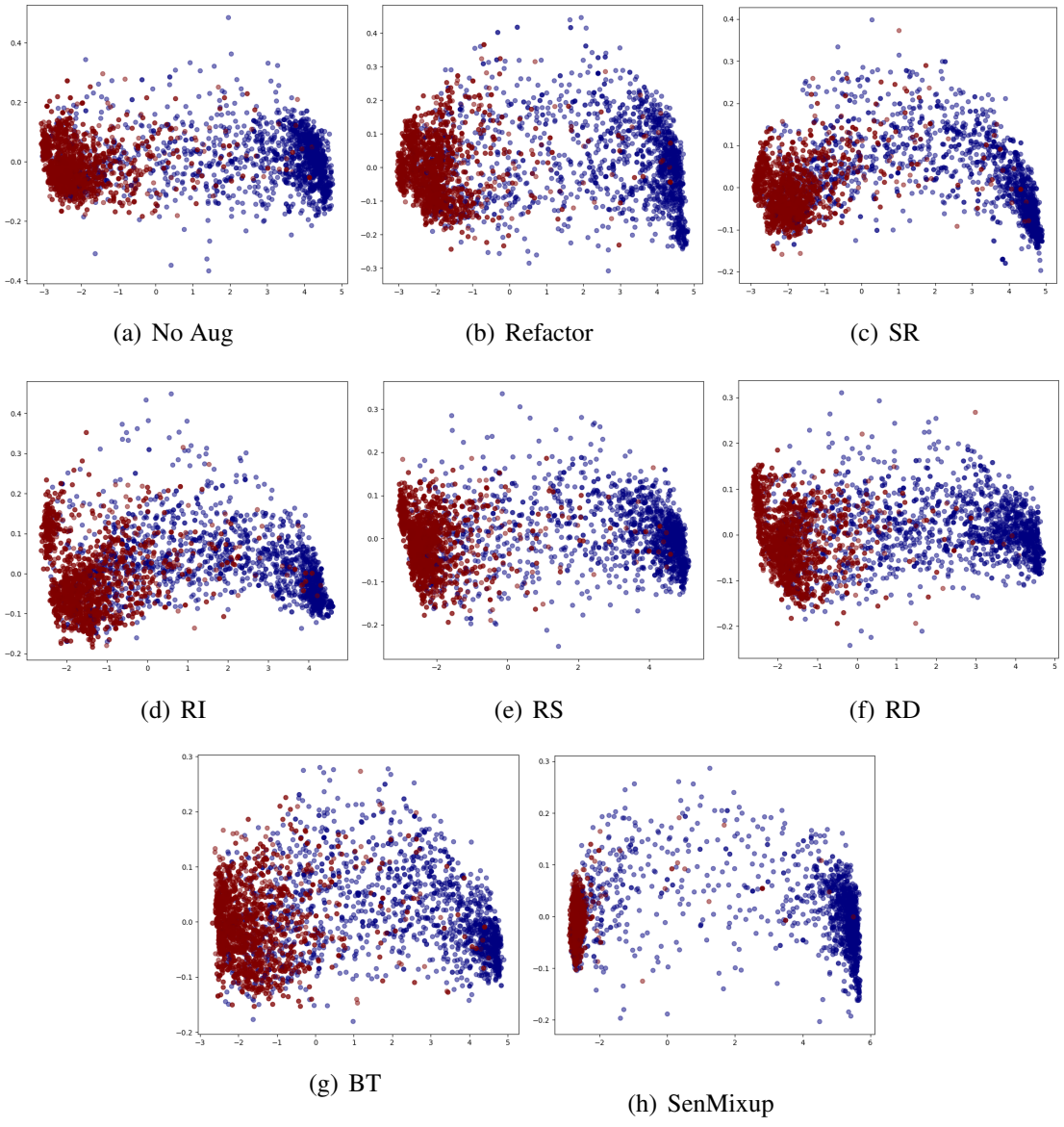
Accuracy. We discuss the accuracy via the visualization of code embeddings. Specifically, 1) we extract code embeddings produced by each trained model using all the test data as code features. 2) Since the extracted features are high-dimensional and challenging to analyze, we reduce the dimensions to two using the *Principal Component Analysis (PCA)* [53] algorithm. 3) Subsequently, we visualize the reduced 2-dimensional vectors. Typically, features extracted by a well-trained model exhibit a clearer distribution based on the data labels.

Fig. 3.6 depicts the visualized features extracted from CodeBERT using the Refactory (Task: Bug detection) dataset. We can see that 1) Data augmentation methods including SR, RI, RS, RD, and *SenMixup* are beneficial in helping accuracy since they significantly improve the representation characteristics in binary classification tasks, making the data features exhibit clear classification trends compared to not using data augmentation. 2) in this case, the code embeddings trained using Mixup-based data augmentation exhibit more evenly distributed class-specific representations compared to embeddings extracted by other data augmentation models. The distinction between buggy and correct codes is pronounced (shown in Fig. 3.6(h)).

Smoother interpolations indicate a smaller volume of hidden representations [76], effectively constraining the number of directions with significant variance. Consequently, through interpolating the hidden space between two different data points, Mixup-based data augmentation methods assist the source code model in capturing the underlying code data structure, enhancing accuracy performance. In summary, these results suggest that Mixup-based data augmentation methods can effectively train high-quality code embeddings, thereby contributing to the development of superior code models for downstream code classification tasks.

Robustness. We discuss the robustness based on the metric *confidence*. Typically, when a DNN model exhibits higher confidence in its predictions on test data, it becomes more challenging to execute adversarial attacks on the model [64].

Table 3.7 presents the results of the mean values of maximum possibilities. In most cases, it is evident that source code models trained using mixup-based data augmentation methods, including WordMixup and SenMixup, demonstrate higher confidence in their predictions on the test data compared to source code models trained using other data augmentation methods. While there are a few cases, such as GCJ and Refactory,



(a) No Aug (b) Refactor (c) SR

(d) RI (e) RS (f) RD

(g) BT (h) SenMixup

Figure 3.6: Visualization of code embeddings after dimension reduction using Principal Component Analysis (PCA). Model: CodeBERT, dataset: Refactory

Table 3.7: Mean values of maximum probabilities. The best and second-best results are highlighted in gray and blue, respectively. Tasks include **Problem Classification** (Java250), **Bug detection** (Refactory), **Authorship Attribution** (GCJ) and **Clone detection** (BigCloneBench)

Model	DA method	GCJ	Refactory	Java250	BigCloneBench
CodeBERT	No Aug	0.0381842	0.9367431	0.9553056	0.9653085
	Refactor	0.0391791	0.9643209	0.9621801	0.9655135
	SR	0.0380601	0.9527591	0.9342893	0.9582997
	RI	0.0371315	0.9189276	0.9328093	0.9705281
	RS	0.0392912	0.9561052	0.9580978	0.9411782
	RD	0.0392801	0.9724103	0.9635246	0.9630691
	BT	0.0379615	0.9566809	0.9418421	0.9667154
	WordMixup	0.0392841	0.9657822	0.9639287	-
	SenMixup	0.0392896	0.9681156	0.9652706	-

where the mean values of maximum probabilities of using *SenMixup* to enhance model training are not the absolute best, it still ranks second, indicating strong performance (e.g., compare GCJ-RS-0.0392912 to GCJ-*SenMixup*-0.0392896).

As two variants of Mixup, the robustness experiments of *SenMixup* and *WordMixup* yield similar conclusions to the usage of Mixup. Adversarial attacks involve subtle alterations to input data to deceive the model's predictions. We can effectively smooth the data distribution in the feature space by taking two code data to generate new data as a linear combination of the original data. The smoothing effect on the decision boundary induced by mixing can diminish the effectiveness of these perturbations on the model's behavior. Furthermore, existing literature [101] has also proven that minimizing the *Mixup* loss approximates the minimization of an upper bound on the adversarial loss. Therefore, as a data-adaptive regularization, Mixup-based data augmentation methods can improve the performance of robustness.

3.4.3 Threats to Validity

The threats to the internal validity of this study stem from the implementation of standard training and data augmentation methods. The code for model training, SenMixup, and WordMixup methods were sourced from official projects. Additionally, the code refactoring methods for Java were adapted from existing works and modified to work

with the Python language [61, 87].

External threats to validity revolve around the chosen code-related tasks, datasets, deep neural networks (DNNs), and data augmentation methods. The study encompasses four code classification tasks: problem classification, bug detection, authorship attribution, and clone detection across six datasets. The analysis employs four types of DNN models, including two widely used pre-trained programming language models. The data augmentation methods include code refactoring techniques, covering common methods in the literature, and data augmentation methods from NLP with a strong citation history.

Construct threats to validity mainly arise from parameters, randomness, and the choice of evaluation measures. Parameters for methods like Mixup and data augmentation techniques from NLP adhere to their original recommendations. To mitigate the influence of randomness, experiments were repeated five times, and the results reported are the averages. The evaluation measures employed in the study encompass accuracy and robustness, with the latter to gauge the generalization ability of the DNNs.

3.5 Conclusion

Our empirical study has shed light on the significance of data augmentation in code learning. It has delivered valuable insights into the efficacy of various augmentation techniques across different downstream tasks and DNN models. In particular, it has been observed that linear interpolation methods, such as *SenMixup* and *Manifold-Mixup*, effectively enhance the accuracy and robustness of most DNNs, excluding pre-trained PL models. Techniques like *random deletion (RD)* and *random swap (RS)* that mildly disrupt the syntax of source code have proven to be especially valuable for pre-trained PL models. Notably, linear interpolation methods exhibit capability even when working with limited training data. These findings open doors further to enhance the effectiveness of data augmentation in code-related tasks, ultimately advancing program comprehension and expediting software development.

Chapter 4

On the Effectiveness of Graph Data Augmentation for Source Code Model

4.1 Introduction

Quality assurance in software systems has long been a critical concern. With the recent surge in machine learning, source code learning, a subfield of*machine learning for big code (ML4Code)* [1], has gained significant attention. Its primary objective is to enhance the reliability of software systems. Source code learning has shown remarkable promise in various downstream software engineering tasks. Examples include code clone detection [1], vulnerability detection [50], and problem classification [62].

The effectiveness of source code learning largely hinges on *code representation*, a fundamental aspect. This process involves creating the representations necessary for downstream tasks, effectively converting raw code into machine-readable vectors. Two main types of code representation techniques have been extensively studied: sequence-level representation and graph-level representation. Sequence-level representation entails transforming source code into a sequence of tokens, encoding each token and the entire program using predefined integers. In contrast, graph-level representation employs graph-structured data, like an *abstract syntax tree (AST)*, enabling the capture of more syntactic and semantic information across the elements in the source code.

These code representation frameworks enable the use of source code data and their corresponding labels in the training of machine learning models. However, data scarcity poses a significant challenge to source code learning. Preparing data, en-

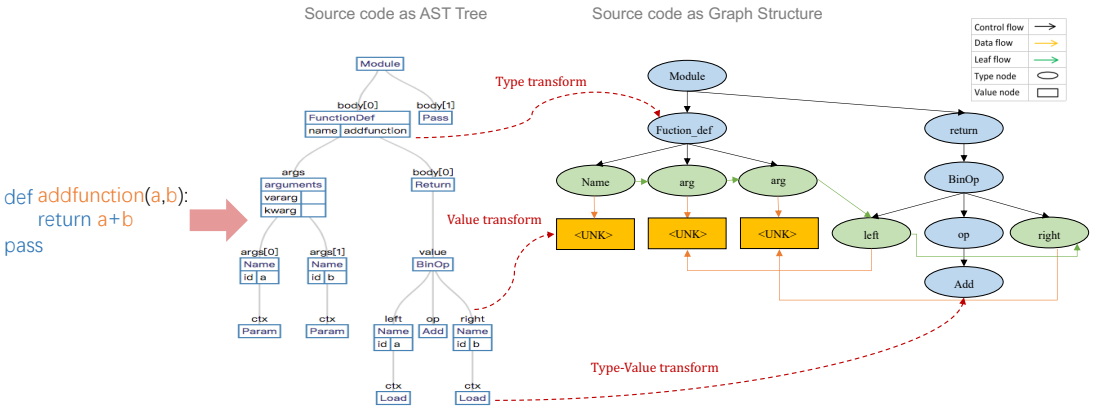


Figure 4.1: An example of source code graph

compassing tasks such as collection, cleaning, and manual labeling, is not only time-consuming but also labor-intensive. Data labeling, in particular, is an expensive and demanding task in source code learning, often requiring specialized expertise. For example, a study by [108] revealed that labeling just four code libraries can consume up to 600 person-hours. Consequently, generating an adequate volume of training data for source code learning remains a daunting challenge and a primary bottleneck affecting its performance.

In various fields of machine learning, including *computer vision (CV)*, practitioners have effectively addressed data scarcity issues by employing *data augmentation*. Data augmentation is a technique that creates additional training data by applying a variety of modifications to existing data while preserving their underlying semantics. For example, in image classification, common data augmentation techniques involve image transformations [29] such as *rotation* and *adding noise* to generate a new set of images for supplementary training data. Recent studies have demonstrated that data augmentation is highly effective in enhancing the accuracy and robustness of machine learning models, as highlighted in works like those by [60, 63].

While data augmentation has gained significant traction in various fields, it has seen limited exploration in the context of source code learning. Currently, only a few works have delved into the data augmentation problem, as exemplified by [2, 97]. These efforts have primarily centered on employing straightforward code refactoring methods like local variable renaming, code duplication, and dead store elimination. However, the results reported in these works suggest that the performance achieved by

Table 4.1: Details of tasks, datasets, and DNN models. **#Training**, **#Test**, and **#Parameters** are the number of data for training, testing, and parameters, respectively.

Dataset	Language	Task	#Training	#Test	Model	#Parameters
Java250	Java	Problem classification	48000	15000	GCN	3.59M
					GCN-Virtual	5.57M
Python800	Python	Problem classification	153600	48000	GIN	5.29M
					GIN-Virtual	6.35M
CodRep1	Java	Bug detection	6944	772		
Refactory	Python	Bug detection	3380	423	GCN	0.45M
					GAT	0.46M
Google Code Jam (GCJ)	Python	Authorship attribution	528	132	GGNN	1.80M
					SAGE	0.90M
BigCloneBench	Java	Clone detection	90102	4000		

these basic methods is not entirely satisfactory. Thus, there remains a need for more effective data augmentation methods in the realm of source code learning to enhance training performance and the overall quality of code-related machine learning models.

Our contributions. It's noteworthy that source code can be effectively represented as graph data structures, and there's a dynamic research field [1] in which the *graph learning* technique is applied to enhance source code learning. Graph learning is a specialized approach designed for graph-structured data, often harnessing graph neural networks (GNNs), which are architectures tailored to process graph data. In the realm of graph learning, a multitude of data augmentation techniques is available, tailored explicitly to graph-structured data. Some prominent methods include *DropNode* and *DropEdge*. These techniques have demonstrated significant success in the context of graph-based machine learning, producing highly accurate and robust GNN models. Importantly, they have the potential to be adapted and applied to code-related tasks. However, the application and performance of these methods in source code learning tasks have yet to be thoroughly investigated and explored. This presents an intriguing avenue for further research in the field.

In this research, considering the graph-like nature of source code, we propose applying data augmentation techniques from graph learning to source code learning tasks. We've undertaken a comprehensive empirical investigation to ascertain whether these novel data augmentation strategies are more effective than the traditional code refactoring methods.

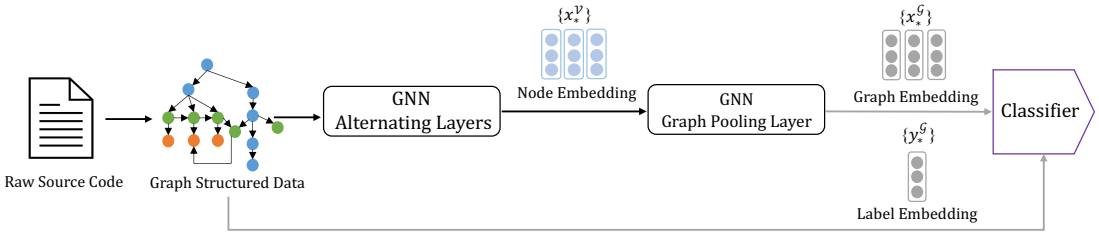


Figure 4.2: Overview of graph classification via GNNs

We designed the large-scale empirical study to answer the following three research questions:

- RQ1: Can existing graph data augmentation methods produce more accurate and robust GNN models than code data augmentation methods?
- RQ2: Why do existing graph data augmentation methods produce more accurate and robust GNN models?
- RQ3: How do existing graph data augmentation methods affect computational resources compared to code data augmentation methods?

Additionally, to comprehensively present the advantages of this study, as described in Table 4.2, we compare our work with existing studies, considering various aspects, including their properties, dependencies, and diversities.

4.2 Methodology

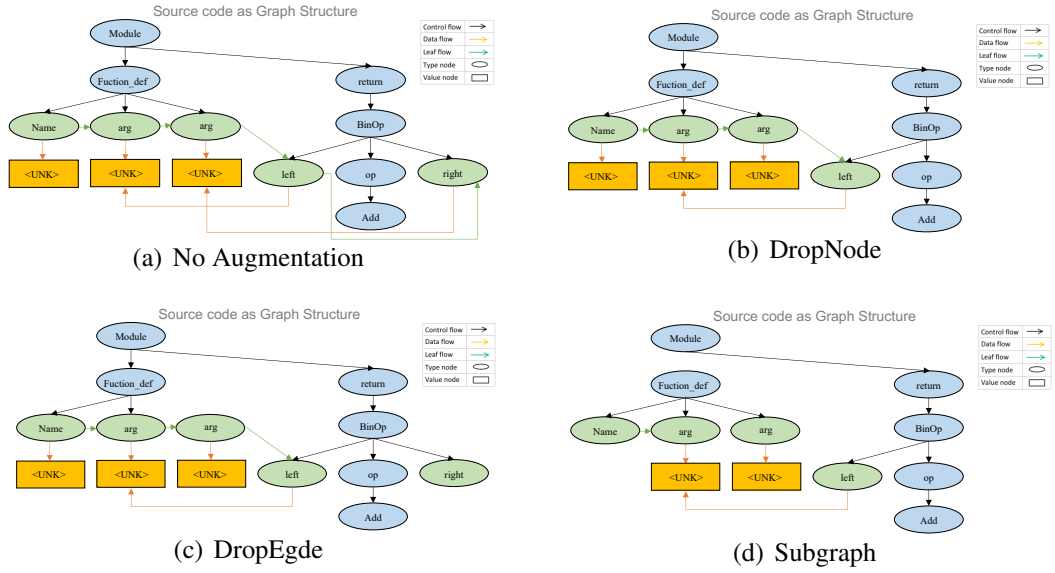
4.2.1 Overview

We first initiated our study by reviewing and categorizing existing data augmentation methods in the field of graph learning as documented in the literature. We then proceeded to adapt four types of data augmentation techniques to the specific context of source code learning. Subsequently, we performed a rigorous performance comparison of these newly adapted data augmentation methods against two baseline approaches: the no-augmentation method and the code refactoring method. Our evaluation encompassed four critical software engineering tasks, and we examined their performance across seven diverse neural network architectures. Through this extensive empirical analysis, we aimed to shed light on the efficacy of these new data augmentation strategies in the realm of source code learning.

Table 4.2: Comparing existing code data augmentation methods by various aspects relating to their properties, dependencies, and diversities. *Transformation* denotes the program transformation method utilized for data augmentation. *Rule* involves modifying the syntactic structure without altering the semantic information. *Mixup* represents a series of Mixup variants designed specifically for code data. *Model* is the method that leverages an *LLM* to generate source code. The term *parsing* means the type of feature utilized by the data augmentation method during program parsing, including features such as *AST*, *CFG*, and *DFG*. *Understanding Task* denotes whether the code data augmentation method is suitable for source code understanding tasks such as program classification, bug detection, and clone detection. Similarly, *Generation Task* means whether the code data augmentation method is applied to generation tasks like program repair and code completion. Finally, *Mutil-languages* stands for whether the code data augmentation can support different programming languages.

Method	**Category**	**Transformation**	**Parsing**	**Understanding Task**	**Generation Task**	**Mutil-lanaguages**
Our work	Empirical	Rule + Mixup	AST + CFG + DFG	✓	✓	✓
MixCode [14]	Technical	Mixup	-	✓		✓
PYBUGLAB [2]	Technical	Rule	AST	✓	✓	
SPAT [97]	Technical	Rule	AST	✓	✓	
Linear Extrapolation [48]	Technical	Mixup	-	✓		✓
Binary Interpolation [48]	Technical	Mixup	-	✓		✓
Search-based Testing [61]	Technical	Rule	AST	✓	✓	
MixCode-Refactor [14]	Technical	Rule	AST	✓	✓	
Mixpooling [13]	Empirical	Mixup	-	✓		✓
CodeT5 [82]	Technical	Model	-	✓	✓	✓
CodeBERT [19]	Technical	Model	-	✓	✓	✓
GraphCodeBERT [25]	Technical	Model	-	✓	✓	✓

In alignment with recent research [105], we've compiled four data augmentation techniques, specifically: *DropNode*, *DropEdge*, *Subgraph*, and *Manifold-Mixup* from the domain of graph learning. These methods, originating from the field of graph learning, have been repurposed and adapted to the context of source code learning. To assess their effectiveness, we have undertaken a comparative analysis with the simple data augmentation techniques that involve code refactoring. To elaborate on our approach, the transformation process is illustrated in Figure 4.1, where the yellow line signifies the flow of data and the black line signifies the flow of control. We employ the *abstract syntax tree (AST)* and various code flows, including control and data flow, to represent source code as graphs. This approach allows us to capture the intricate relationships between the different components of the code. Utilizing this graph-based structure, we apply these graph-learning data augmentation methods to the domain of



(a) No Augmentation (b) DropNode

(c) DropEgde (d) Subgraph

Figure 4.3: Examples of data augmentation methods from graph learning

source code learning for a comprehensive evaluation.

Figure 4.2 provides an overview of how a Graph Neural Network (GNN) is employed to handle classification tasks with graph-structured data. Initially, the GNN takes the graph-structured data, parsed from the source code, as input and transforms it into node embeddings, representing a formalism, using alternating layers. These layers capture and learn information from the graph nodes. Once the embeddings for the graph nodes are learned, graph pooling layers are utilized to extract structural information from the graph. This is based on the knowledge acquired from the alternating layers.

4.2.2 Dropout-Based Methods

These data augmentation methods fall under a category that leverages a regularization technique known as *Dropout* [70], which originated in the field of graph learning. *Overfitting* [78] is a well-known issue in deep learning, where a model becomes excessively specialized in the training data, essentially memorizing noise and random variations rather than capturing meaningful underlying features. This leads to models that perform very well on the training data but struggle when presented with unseen data.

Dropout is a computationally efficient and highly effective regularization method with widespread application in deep neural networks to mitigate overfitting. Dropout introduces randomness and redundancy into the network during training. The technique is quite simple in concept. It generates multiple network variations within a single model by randomly deactivating neural nodes during training. Specifically, during each training iteration, Dropout randomly sets a portion of neurons within a neural network layer to zero, meaning that these neurons don't participate in the forward or backward pass for that iteration. Since the network doesn't know which neurons will be dropped in advance, it adapts by creating more robust and general features that are less dependent on the presence of any specific neuron. This encourages the network to learn valuable features across different dataset parts, improving generalization [4]. In our context, we utilize two variants of Dropout designed for processing graph-structured data, known as *DropNode* and *DropEdge*. These methods aim to introduce a degree of randomness into the graph learning process for source code data.

DropNode. Like the Dropout technique introduced in the context of deep learning by [70], DropNode presents a comparable approach in graph learning. DropNode aims to mitigate overfitting during the training of GNNs by creating numerous variations of the original graph. This is achieved by randomly removing a set of nodes from the underlying graph and associated edges. It's important to note that DropNode operates at the data level rather than directly modifying the neural units within the network architecture.

In our study, we leverage this DropNode method for data augmentation. Initially, we convert the source code into a graph-structured representation. We then apply the DropNode technique to generate variants of the original graphs, which serve as additional training data. Importantly, our node-dropping operation ensures that the absence of certain nodes does not compromise the semantics of the original graph. This constraint aligns with the recommendation put forward by [18].

Given a graph data $\mathcal{G}(\mathcal{V}, \varepsilon)$, where $\mathcal{V}$ is the vertex (node) set, ε means the edge set and $\mathcal{M}$ represents the vertex (node) set that we drop. Based on Eq. (2.1) (as discussed in Section 2), we simply define the graph representation using DropNode as follows:

$$\mathbf{H}_u^k = \mathbf{GNN}\left(\mathbf{H}_u^{k-1}, W, \sum_{v \in \left(\mathcal{N}(u) \setminus \mathcal{M}\right)} \mathbf{H}_v^{k-1}\right) \tag{4.1}$$

To better understand, we give an example to present how DropNode works. As

shown in Figure 4.3(b), we randomly select a node and delete it along with its connections to other nodes, which amounts to a probability $p = 0.1$ of the total number of nodes.

DropEdge. In a manner akin to DropNode, *DropEdge* is another adaptation of the Dropout technique specifically tailored for GNNs. However, DropEdge operates on graph edges instead of manipulating node features, introducing variability into the original graph during training. With DropEdge, each edge in the graph is probabilistically set to zero with a designated probability, denoted as p, during each training iteration. This process effectively deletes specific connections, contributing to the GNN's resilience in the face of noisy or incomplete information encountered during training. Furthermore, removing edges introduces controlled noise into the GNN's message-passing mechanism. This noise is a regularization method, mimicking real-world uncertainties by randomly removing edges during training. This aids the GNN in becoming more robust to fluctuations in the data. DropEdge, as a whole, enhances the generalization capabilities of GNNs and mitigates common issues like over-smoothing and overfitting. This is especially beneficial for deeper GNN architectures [66].

Given a graph data $\mathcal{G}(\mathcal{V}, \varepsilon)$ and its transformation by DropEdge $\mathcal{G}(\mathcal{V}, \varepsilon^*)$, where ε^* means the new edge set after edge removing operator. For the node u, and one of its neighbor nodes v, uv ($uv \in \varepsilon, uv \notin \varepsilon^*$) represents the dropped edge. Based on Eq. (2.1), the graph representation using DropEdge can be written as follows:

$$\mathbf{H}_u^k = \mathbf{GNN}\left(\mathbf{H}_u^{k-1}, W, \sum_{v \in \mathcal{N}(u), (uv \in \varepsilon, uv \notin \varepsilon^*)} \mathbf{H}_v^{k-1}\right) \tag{4.2}$$

Furthermore, we provide an example, as shown in Figure 4.3(c). We randomly select several edges equivalent to a probability of $p = 0.1$ of the total number of edges. Then, we remove the information associated with these edges chosen during the GNN training phase.

4.2.3 Sampling-Based Methods

In the realm of data augmentation, sampling-based methods are often employed to generate new synthetic data by selecting data points from an existing dataset. These techniques have proven valuable for data augmentation in various applications [17]. Specifically, in the context of graph data augmentation, we introduce two widely-used methods as follows:

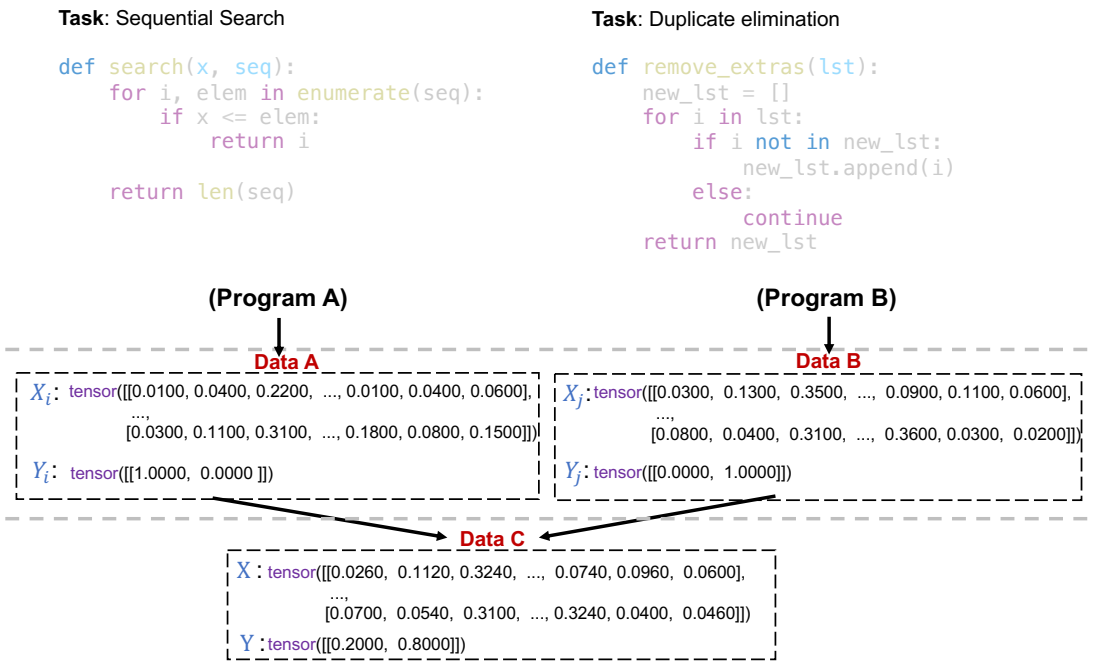


Figure 4.4: An example of linear interpolation of two programs, Dataset:Refactory

Subgraph. In contrast to the *DropNode* and *DropEdge*methods, this approach generates augmented graph data by employing a technique known as *Subgraph* [83]. Using a random walk, the *Subgraph* method selects a connected *Subgraph* from the original graph. Like the dropout-based techniques, this method is designed to preserve a portion of the semantic information from the original graph while enhancing it. *Subgraph* operates by selecting relevant features from a larger graph, effectively reducing the dataset size compared to the original graph. This reduction in dataset size can combat overfitting by limiting the number of features used during model training. Moreover, *Subgraph* is valuable for reducing data noise since it focuses on specific substructures or relevant graph sections. Noise in the data can contribute to overfitting, and therefore, employing the *Subgraph* method can be an effective strategy for mitigating this problem and improving the overall performance of the GNN model.

Random walk is a widely used sampling technique in GNNs. It is employed to produce stochastically and sample nodes and edges from a graph. To explain this further, consider the example illustrated in Figure 4.3(d). The process begins by randomly selecting a node within the graph as the starting point. Subsequently, a probability denoted as p is used to choose neighboring nodes to move to in the next step. This

process is repeated for a predetermined number of steps. Consequently, the nodes encountered during this random walk are sampled by the graph's inherent structure and connectivity. Random walk is a valuable technique for generating samples that capture the graph's structural and relational aspects.

Manifold-Mixup. *Mixup* [99] is an advanced data augmentation method that finds wide application in image data classification tasks due to its demonstrated effectiveness in improving the accuracy and robustness of DNNs [101]. Specifically, the Mixup method randomly selects two raw images from the training data. It then linearly combines the features and labels of these two chosen images to generate a new image and label, which are treated as augmented training data. Mixup's key function is smoothing the data distribution within the feature space by taking two data points and creating a new data point as a linear combination of the original ones. This smoothing process effectively reduces the sharpness of decision boundaries between different classes, making them less prone to overfitting [41]. Moreover, Mixup has demonstrated its ability to enhance model robustness, particularly when dealing with adversarial attacks. Adversarial attacks involve making subtle input data changes to mislead the model's predictions. The smoothing effect on the decision boundary, brought about by the mixing of data, reduces the effectiveness of these perturbations in causing the model to make incorrect predictions [101].

As a variant of *Mixup*, *Manifold-Mixup* [76, 84] is specially designed for graph data classification by interpolating graph-level embedding. Technically, given two pairs $\left(x_i^{\mathcal{G}}, y_i^{\mathcal{G}}\right)$ and $\left(x_j^{\mathcal{G}}, y_j^{\mathcal{G}}\right)$ of graph data and their labels, the interpolated graph data pair $\left(x_{mix}^{\mathcal{G}}, y_{mix}^{\mathcal{G}}\right)$ is calculated by:

$$\begin{aligned} x_{mix}^{\mathcal{G}} &= \lambda x_i^{\mathcal{G}} + (1-\lambda) x_j^{\mathcal{G}} \\ y_{mix}^{\mathcal{G}} &= \lambda y_i^{\mathcal{G}} + (1-\lambda) y_j^{\mathcal{G}} \end{aligned} \tag{4.3}$$

where, $x^{\mathcal{G}}$ is the graph embedding and $y^{\mathcal{G}}$ means its label embedding with one-hot embedding, and λ is the *Mixup* ratio sampled from a Beta distribution with a shape parameter α $\left(\lambda \sim Beta\left(\alpha, \alpha\right)\right)$, as done in [99].

To better understand how the linear interpolation method works, we use an example to illustrate the details of the process of linearly mixing two different programs. (Program A and B from dataset Refactory with label 0 and label 1, respectively) as

depicted in Figure 4.4. First, we convert two different programs into the vector space using *GCN* [42]. Additionally, we encode their corresponding labels using one-hot encoding, representing them with two classes (see Data A (X_i, Y_i) and Data B $\left(X_j, Y_j\right)$ in Figure 4.4). Subsequently, we perform linear mixing on the code vectors and label vectors of Data A and Data B, thereby creating augmented training data (X, Y) that is treated as the augmented training data to be fed into the model (see Data C in Figure 4.4). In this example, we set $\lambda = 0.2$ in Eq. (4.3) to conduct linear interpolation of a pair of programs.

4.3 Evaluation

In our study, we investigate two prominent programming languages, Java and Python, and emphasize four pivotal downstream tasks in source code classification: problem classification, bug detection, authorship attribution, and clone detection. We evaluate these tasks across seven distinct Graph Neural Network (GNN) model architectures, including GCN, GIN, GCN-Virtual, GIN-Virtual, GAT, GGNN, and SAGE. For a comprehensive understanding of the datasets and models considered in our experiments, please refer to Table 4.1. We employ well-established source code classification tasks and datasets commonly used in source code learning as the basis for our experiments and evaluations.

4.3.1 RQ1: Can existing graph data augmentation methods produce more accurate and robust GNN models than code data augmentation methods?

Study Design. In this research question, our approach begins with preparing the initial training data and the random initialization of source code GNN models. These models are then trained using various graph data augmentation techniques. Ultimately, we measure the final accuracy of the GNNs after a set number of training epochs. Our experiment encompasses two main evaluation aspects: *accuracy* and *evasion attack* (which will be introduced later). Clean Accuracy assesses how well the GNN models perform on the original test data. This test data generally adheres to the same data distribution as the training data. Clean accuracy calculates the percentage of correctly

classified data within the entire test dataset. It quantifies the model's ability to make correct predictions on clean, unaltered data. Evasion Attack is another evaluation aspect that will be discussed in more detail later. It measures how robust the model is against adversarial attacks, which are attempts to manipulate or deceive the model's predictions through carefully crafted input data.

$$Accuracy = \frac{\#Corrected\,classified\,data}{\#Entire\,data}\% \tag{4.4}$$

Additionally, our study significantly emphasizes evaluating the adversarial robustness of the trained GNN models. In this context, robustness signifies the ability of GNN models to generalize effectively to handle data that originates from unseen data distributions, for example, graph data with noise or other variations. Previous research in other domains [63] has demonstrated that data augmentation can enhance the adversarial robustness of Deep Neural Network (DNN) models. To explore whether these findings can be extended to GNN models for code, we employ a widely recognized adversarial attack method designed explicitly for GNNs. This method is referred to as *Projected Randomized Block Coordinate Descent (PRBCD)* [21]. PRBCD is utilized in our test data to assess the robustness of the GNN code models trained using various graph data augmentation methods. It does so efficiently by leveraging gradient information and relaxing the nature of the adjacency matrix $\{0,1\}$ to $[0,1]$ while not directly altering the node features. In our evasion attack experiments, robust test accuracy means the accuracy of the trained GNN models on the robust test data, which is generated by *PRBCD*.

$$Robustness = \frac{\#Corrected\,classified\,data}{\#Entire\,robust\,data}\% \tag{4.5}$$

Furthermore, we totally collect code refactoring methods as our baseline approaches from the existing study [14]. All program transformation methods can be found on their project site [4].

Results Analysis. The results displayed in Table 4.5 clearly highlight the impact of different data augmentation methods on the accuracy of GNN models for code classification tasks. For GCN, *Manifold-Mixup* outperforms other data augmentation methods, showing the best performance in four out of six datasets. It leads to a maximum accuracy improvement of 1.60% and an average improvement of 0.94% compared to *No Aug*. For GAT, GGNN, and SAGE, similar trends are observed with these models.

Table 4.3: A case study of mean values of maximum probabilities (well-trained). The best and second-best results are highlighted in gray and blue, respectively.

Model	DA method	GCJ	Refactory	Java250	BigCloneBench
GCN	No Aug	0.0217816	0.7837402	0.9127041	0.9211723
	Refactor	0.0220143	0.7915629	0.9101877	0.9205861
	DropNode	0.0226572	0.8042617	0.9083258	0.9371285
	DropEdge	0.0203749	0.7601385	0.8936216	0.9398362
	Subgraph	0.0195256	0.7783275	0.9181428	0.9186201
	Manifold-Mixup	0.0239415	0.7985071	0.9201572	0.9385479

Manifold-Mixup consistently demonstrates its exceptional ability to improve accuracy, prevailing in 9 out of 12 cases. For GGNN, it outperforms other augmentation methods in four datasets. For GIN, GCN-Virtual, and GIN-Virtual, DropEdge shines in these cases, achieving the best performance in three out of six scenarios. Manifold-Mixup performs exceptionally well in a single case, indicating that it may not be ideal for these specific GNN architectures. Interestingly, Refactor doesn't consistently enhance GNN model performance. In one case (BigCloneBench-GAT), it results in a 0.76% decrease in accuracy compared to no data augmentation. On the other hand, Manifold-Mixup consistently delivers accuracy improvements, with a maximum improvement of 1.53% and an average improvement of 0.90%. In conclusion, these results indicate that, in the context of using GNN models for code feature learning, Manifold-Mixup stands out as the top choice for augmenting source code datasets. In most cases, it exhibits the best accuracy improvement, surpassing other augmentation techniques. Manifold-Mixup particularly excels in the Refactory, CodRep1, GCJ, and BigCloneBench datasets compared to Refactor. It is a highly effective augmentation method for code data when working with GNN models, consistently enhancing their accuracy in various code classification tasks.

Table 4.6 shows the evaluation of the adversarial robustness of GNN models using the PRBCD attack method on six datasets and provides some interesting insights. For GCN, Manifold-Mixup emerges as the top choice, leading to the highest robust test accuracy in three out of six datasets. In the GCJ dataset, Manifold-Mixup achieves a substantial improvement of up to 2.75% compared to training without data augmentation. For GAT, GGNN, SAGE, and GIN-Virtual, Manifold-Mixup continues to showcase its exceptional ability to improve robust test accuracy across these models, with

Table 4.4: A case study of mean values of maximum probabilities (10^{th} epoch). The best and second-best results are highlighted in gray and blue, respectively.

Model	DA method	GCJ	Refactory	Java250	BigCloneBench
GCN	No Aug	0.0173681	0.5351947	0.8279421	0.5589432
	Refactor	0.0174542	0.5382019	0.8201596	0.5503627
	DropNode	0.0172365	0.5397028	0.8314702	0.5578473
	DropEdge	0.0168145	0.5401587	0.8215417	0.5591324
	Subgraph	0.0178708	0.5251761	0.8047591	0.5487044
	Manifold-Mixup	0.0170125	0.5317519	0.8357413	0.5579326

support from 11 out of 14 cases. Interestingly, DropNode is the most effective choice for GIN and GCN-Virtual, achieving a maximum improvement of 1.53% compared to training without data augmentation. A notable trend in these results is that Manifold-Mixup tends to be the most effective in enhancing robust test accuracy, particularly as the dataset size decreases. For instance, in the GCJ-GGNN case, Manifold-Mixup outperforms training without data augmentation by up to 4.09%. Additionally, it shows an advantage over Refactor of up to 3.73% in the Refactory-SAGE case. However, it's worth noting that for sparser datasets like CodRep1 and GCJ, the graph data augmentation method DropEdge doesn't perform as effectively. Only 3 out of 8 cases show that *DropEdge* can improve the robust test accuracy compared to training without data augmentation. This phenomenon could be attributed to the over-sparcity of the adjacency matrix generated by DropEdge, potentially leading to the loss of critical structural information for node representation. This aligns with previous findings [38], suggesting that DropEdge may be unreliable for sparse graphs. In summary, graph data augmentation methods, in general, provide limited benefits in enhancing the adversarial robustness of source code models. Furthermore, no single method consistently improves model robustness across different datasets and models compared to not using data augmentation. The choice of augmentation method should be tailored to the specific dataset and model architecture to achieve the desired results in terms of robustness.

Table 4.5: Effectiveness of data augmentation methods w.r.t. test accuracy ↑ (average ± standard deviation, %) on original test data. **No Aug**: without data augmentation. The best results are highlighted in gray. The tested DNN models are GNNs.

Dataset	DA method	Model: GCN	GIN	GCN-Virtual	GIN-Virtual
Java250	No Aug (Baseline)	92.22 ± 0.11	92.47 ± 0.16	93.42 ± 0.28	91.48 ± 0.62
	Refactor	92.31 ± 0.41	92.24 ± 0.56	93.56 ± 0.36	92.34 ± 0.56
	Manifold-Mixup	92.12 ± 0.33	93.41 ± 0.39	93.29 ± 0.32	92.67 ± 0.38
	Subgraph	92.95 ± 0.22	92.74 ± 0.55	92.25 ± 0.22	92.75 ± 0.11
	DropNode	92.19 ± 0.41	92.49 ± 0.36	93.21 ± 0.42	92.32 ± 0.51
	DropEdge	92.37 ± 0.34	93.12 ± 0.52	93.66 ± 0.54	91.39 ± 0.45

Dataset	DA method	Model: GCN	GIN	GCN-Virtual	GIN-Virtual
Python800	No Aug (Baseline)	93.21 ± 0.14	93.62 ± 0.02	93.48 ± 0.36	94.08 ± 0.51
	Refactor	93.33 ± 0.46	93.32 ± 0.62	93.53 ± 0.47	94.12 ± 0.46
	Manifold-Mixup	94.06 ± 0.28	93.56 ± 0.33	93.28 ± 0.28	94.49 ± 0.39
	Subgraph	94.01 ± 0.11	93.88 ± 0.45	93.84 ± 0.19	94.21 ± 0.21
	DropNode	93.24 ± 0.45	93.86 ± 0.73	93.85 ± 0.35	93.76 ± 0.26
	DropEdge	93.22 ± 0.39	94.22 ± 0.66	93.43 ± 0.53	94.88 ± 0.34

Dataset	DA method	Model: GCN	GAT	GGNN	SAGE
Refactory	No Aug (Baseline)	80.14 ± 0.21	79.77 ± 0.16	81.94 ± 0.18	80.05 ± 0.19
	Refactor	80.21 ± 0.48	79.87 ± 0.31	81.12 ± 0.49	80.04 ± 0.32
	Manifold-Mixup	81.74 ± 0.22	80.01 ± 0.24	82.14 ± 0.23	79.83 ± 0.12
	Subgraph	80.08 ± 0.12	79.79 ± 0.26	81.13 ± 0.27	79.64 ± 0.24
	DropNode	80.53 ± 0.31	80.02 ± 0.32	82.09 ± 0.31	80.11 ± 0.22
	DropEdge	80.66 ± 0.27	79.49 ± 0.23	82.11 ± 0.21	80.38 ± 0.25

Dataset	DA method	Model: GCN	GAT	GGNN	SAGE
CodRep1	No Aug (Baseline)	61.38 ± 0.23	61.22 ± 0.23	62.63 ± 0.29	62.34 ± 0.16
	Refactor	61.41 ± 0.45	61.12 ± 0.37	62.75 ± 0.21	62.36 ± 0.29
	Manifold-Mixup	62.47 ± 0.13	62.37 ± 0.32	63.71 ± 0.19	62.89 ± 0.18
	Subgraph	61.04 ± 0.21	61.08 ± 0.14	62.65 ± 0.15	62.98 ± 0.13
	DropNode	61.61 ± 0.19	61.37 ± 0.22	62.59 ± 0.27	61.77 ± 0.19
	DropEdge	61.43 ± 0.23	60.87 ± 0.18	62.21 ± 0.25	62.36 ± 0.25

Dataset	DA method	Model: GCN	GAT	GGNN	SAGE
GCJ	No Aug (Baseline)	52.65 ± 0.13	53.77 ± 0.11	54.88 ± 0.12	52.43 ± 0.15
	Refactor	52.57 ± 0.39	53.87 ± 0.21	54.92 ± 0.23	52.48 ± 0.31
	Manifold-Mixup	52.86 ± 0.43	54.02 ± 0.38	56.03 ± 0.44	52.98 ± 0.44
	Subgraph	52.68 ± 0.47	53.86 ± 0.39	54.69 ± 0.25	52.14 ± 0.39
	DropNode	52.54 ± 0.45	53.81 ± 0.12	55.26 ± 0.29	52.81 ± 0.43
	DropEdge	52.09 ± 0.32	53.11 ± 0.25	55.22 ± 0.21	52.28 ± 0.36

Dataset	DA method	Model: GCN	GAT	GGNN	SAGE
BigCloneBench	No Aug (Baseline)	93.24 ± 0.44	92.43 ± 0.62	94.56 ± 0.52	93.16 ± 0.65
	Refactor	93.51 ± 0.56	91.67 ± 0.73	94.73 ± 0.56	92.83 ± 0.67
	Manifold-Mixup	93.05 ± 0.72	93.14 ± 0.56	95.45 ± 0.64	93.57 ± 0.52
	Subgraph	93.48 ± 0.52	92.74 ± 0.47	94.68 ± 0.59	93.13 ± 0.76
	DropNode	92.14 ± 0.58	92.51 ± 0.43	94.36 ± 0.67	93.24 ± 0.47
	DropEdge	93.67 ± 0.63	92.23 ± 0.51	95.21 ± 0.52	93.45 ± 0.53

4.3.2 RQ2: Why do existing graph data augmentation methods produce more accurate and robust GNN models?

Study Design. This research question investigates why model training with graph data augmentation methods can achieve higher accuracy than other approaches. We

Table 4.6: Effectiveness of data augmentation methods w.r.t. robust test accuracy ↑ (average ± standard deviation, %) on robust test data. **No Aug**: without data augmentation. The best results are highlighted in gray. The tested DNN models are GNNs. The adversarial attack method is PRBCD.

Dataset	DA method	Model			
		GCN	GIN	GCN-Virtual	GIN-Virtual
Java250	No Aug (Baseline)	78.43 ± 2.34	78.87 ± 3.02	77.62 ± 1.35	77.81 ± 1.83
	Refactor	79.14 ± 1.56	77.43 ± 2.04	76.78 ± 2.12	77.14 ± 1.98
	Manifold-Mixup	80.11 ± 1.41	80.47 ± 1.87	78.13 ± 2.01	77.71 ± 2.21
	Subgraph	79.47 ± 3.02	80.13 ± 2.87	78.01 ± 1.58	78.11 ± 3.87
	DropNode	78.07 ± 3.11	77.97 ± 2.45	76.11 ± 1.81	76.42 ± 2.43
	DropEdge	77.79 ± 2.45	79.63 ± 3.31	78.17 ± 1.87	78.07 ± 2.82

Dataset	DA method	Model			
		GCN	GIN	GCN-Virtual	GIN-Virtual
Python800	No Aug (Baseline)	79.51 ± 2.56	80.01 ± 2.11	79.97 ± 1.45	80.14 ± 1.39
	Refactor	78.27 ± 1.67	78.74 ± 2.98	78.11 ± 2.61	81.04 ± 1.85
	Manifold-Mixup	79.67 ± 3.11	80.14 ± 2.54	80.07 ± 1.76	81.57 ± 2.43
	Subgraph	79.42 ± 1.35	81.23 ± 1.76	81.13 ± 2.06	80.46 ± 2.15
	DropNode	80.74 ± 1.62	81.54 ± 2.65	81.47 ± 2.19	80.97 ± 1.88
	DropEdge	80.82 ± 2.31	80.46 ± 2.15	78.49 ± 1.87	81.27 ± 1.52

Dataset	DA method	Model			
		GCN	GAT	GGNN	SAGE
Refactory	No Aug (Baseline)	65.18 ± 2.54	65.58 ± 1.78	66.51 ± 2.55	66.24 ± 3.41
	Refactor	65.78 ± 2.78	64.12 ± 2.56	65.11 ± 1.67	63.14 ± 3.02
	Manifold-Mixup	66.14 ± 3.13	65.86 ± 2.11	67.03 ± 2.45	66.87 ± 2.56
	Subgraph	64.11 ± 3.01	64.87 ± 2.79	66.78 ± 1.67	62.67 ± 2.45
	DropNode	66.57 ± 1.83	63.11 ± 2.87	62.11 ± 3.31	64.42 ± 2.67
	DropEdge	62.07 ± 2.72	63.03 ± 2.13	63.48 ± 2.57	63.87 ± 2.15

Dataset	DA method	Model			
		GCN	GAT	GGNN	SAGE
CodRep1	No Aug (Baseline)	45.51 ± 2.56	45.87 ± 3.13	46.04 ± 3.57	46.28 ± 3.35
	Refactor	45.78 ± 2.01	44.11 ± 2.89	46.37 ± 3.11	45.07 ± 2.92
	Manifold-Mixup	46.87 ± 2.56	46.07 ± 4.02	48.82 ± 2.98	47.11 ± 3.86
	Subgraph	42.17 ± 3.46	41.47 ± 3.43	40.44 ± 4.21	40.37 ± 3.56
	DropNode	43.52 ± 2.43	46.18 ± 2.66	44.98 ± 3.45	43.61 ± 3.56
	DropEdge	41.67 ± 3.35	41.87 ± 3.52	42.71 ± 3.12	43.02 ± 3.33

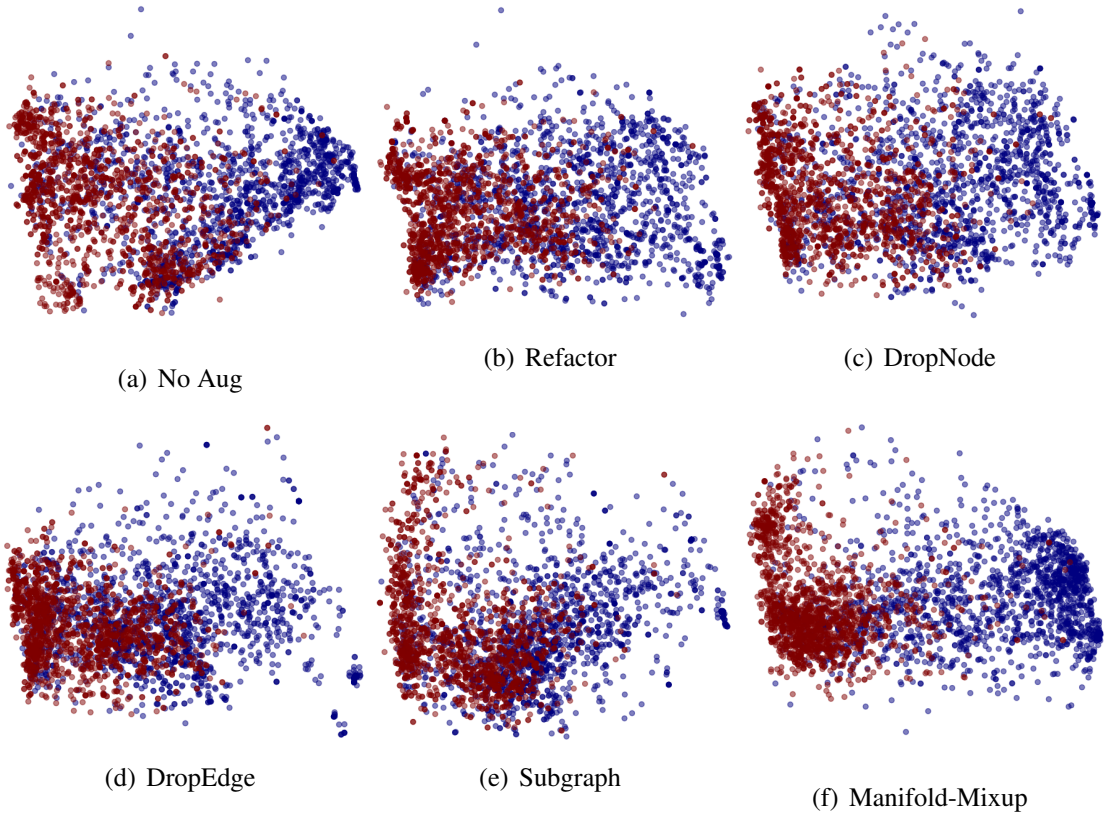
Dataset	DA method	Model			
		GCN	GAT	GGNN	SAGE
GCJ	No Aug (Baseline)	30.14 ± 2.45	30.51 ± 3.53	30.33 ± 3.11	31.07 ± 2.83
	Refactor	30.28 ± 3.56	30.44 ± 3.23	30.87 ± 2.63	31.43 ± 4.19
	Manifold-Mixup	32.89 ± 3.86	33.93 ± 2.88	34.42 ± 3.43	33.68 ± 4.42
	Subgraph	28.72 ± 2.75	28.92 ± 3.92	31.01 ± 2.51	30.11 ± 2.91
	DropNode	30.71 ± 3.44	29.89 ± 3.23	30.15 ± 3.21	31.61 ± 3.39
	DropEdge	29.78 ± 1.52	29.27 ± 2.54	30.87 ± 2.11	30.74 ± 4.11

Dataset	DA method	Model			
		GCN	GAT	GGNN	SAGE
BigCloneBench	No Aug (Baseline)	81.42 ± 1.34	81.81 ± 2.45	82.01 ± 3.67	81.76 ± 2.61
	Refactor	81.58 ± 1.64	81.78 ± 1.58	82.42 ± 2.06	81.69 ± 1.88
	Manifold-Mixup	81.76 ± 2.21	82.11 ± 2.36	82.66 ± 1.49	81.77 ± 3.02
	Subgraph	81.24 ± 1.42	81.19 ± 2.17	82.05 ± 1.91	81.81 ± 1.28
	DropNode	81.88 ± 2.21	82.07 ± 1.21	82.28 ± 3.17	80.89 ± 2.13
	DropEdge	82.14 ± 2.16	81.83 ± 4.01	81.49 ± 2.11	81.87 ± 2.38

employ a multi-step analysis to gain insights into the phenomenon: We begin by extracting code embeddings generated by the trained models using different graph data augmentation methods, including code refactoring and training without data augmentation. We utilize the entire test dataset as the source of code features. 2) Since the ex-

tracted features are typically high-dimensional, which can make analysis complex, we employ *Principal Component Analysis (PCA)* [53], a well-established dimensionality reduction technique, to project these features into a two-dimensional space. 3) We create visual representations With the two-dimensional feature vectors obtained through PCA. Generally, well-trained models exhibit more distinct feature distributions based on data labels. Visualization helps us gain insights into the structure of the learned representations. To further comprehend why graph data augmentation methods, including code refactoring, produce more robust source code models, we evaluate these models' *confidence* when making predictions on the original test data. This analysis involves gathering the maximum probability scores associated with correctly classified test data produced by trained source code models using various data augmentation methods, including code refactoring. We then calculate the mean values of these maximum probability scores. This analysis provides an understanding of the confidence level in the model's predictions when utilizing different data augmentation methods. Furthermore, we present the results for specific datasets related to four crucial source code classification tasks: Java250 for program classification, Refactory for bug detection, GCJ for authorship attribution, and BigCloneBench for clone detection. This approach allows us to provide insights into the impact of data augmentation on various code-related tasks.

Results Analysis. Figure 4.5 offers a visual representation of features extracted from the GCN model using the Refactory dataset for bug detection. Figure 4.5(a) represents the results of model training without data augmentation, while Figure 4.5(f) illustrates the outcomes of model training with the *Manifold-Mixup* data augmentation method. Several insights can be drawn from these visualizations: 1) The benefit of Data Augmentation: Data augmentation methods, including code refactoring, DropNode, and DropEdge, appear advantageous in improving the accuracy of binary classification tasks. These methods lead to distinctive feature distributions that exhibit clear trends for classification, in contrast to the scenario where no data augmentation is used. 2) In the case of subgraph-based data augmentation, the resulting features seem to be sparsely distributed in the two-dimensional vector space. This sparsity indicates a reduction in the dimensionality of hidden representations. However, while this might reduce overfitting, it could also limit the model's ability to effectively capture the underlying data structure. Features generated by the *Manifold-Mixup* method display a more evenly distributed representation specific to each class. This smoother distribu-



(a) No Aug (b) Refactor (c) DropNode

(d) DropEdge (e) Subgraph (f) Manifold-Mixup

Figure 4.5: A case study of visualization of code embeddings after dimension reduction using Principal Component Analysis (PCA).

tion implies a smaller volume of hidden representations [76]. A smaller volume of hidden representations effectively constrains the number of directions with significant variance. Consequently, *Manifold-Mixup* aids the GNN model in capturing the underlying data structure, contributing to enhanced accuracy. These visualizations support the idea that data augmentation methods, particularly *Manifold-Mixup*, play a crucial role in enhancing model performance by improving feature representations in binary classification tasks.

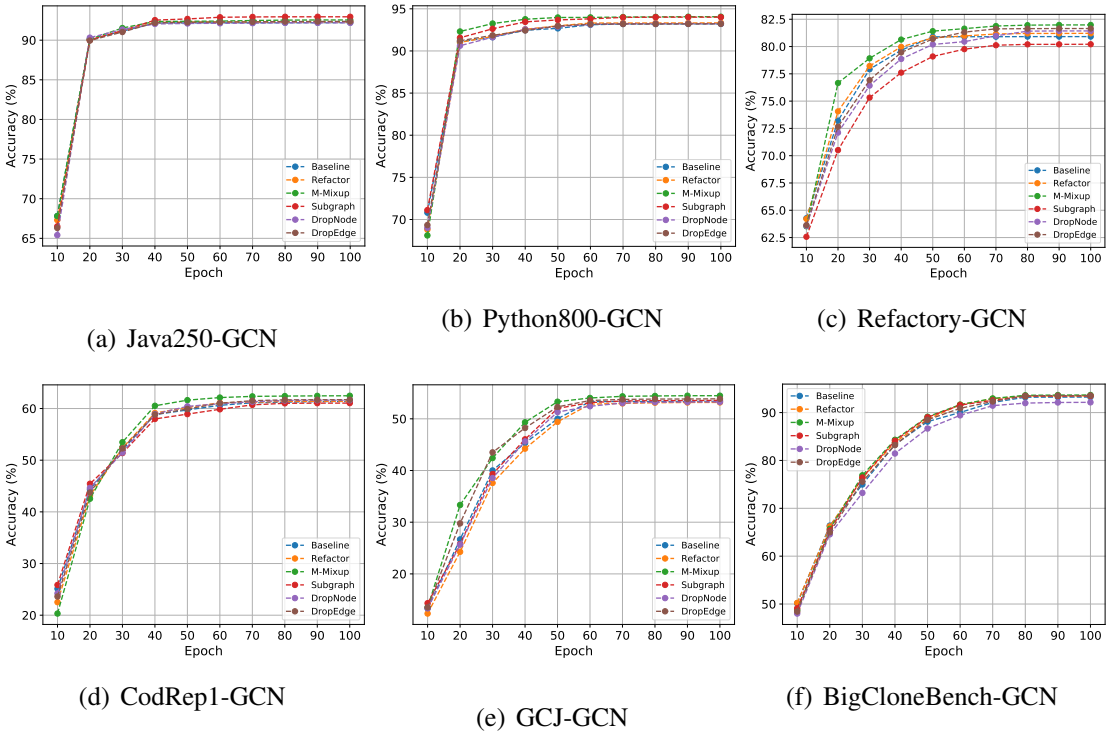
Table 4.3 presents the mean values of maximum probabilities, which reflect the confidence of DNN models in their predictions on test data. Higher confidence in predictions generally makes it more challenging to perform successful adversarial attacks on a model. The results are pretty compelling. In most cases, models trained with *Manifold-Mixup* exhibit higher confidence in their test data predictions than mod-

els trained with other data augmentation methods. While there are a few exceptional cases, such as Refactory and BigCloneBench, where *Manifold-Mixup* does not achieve the absolute best mean values of maximum probabilities, it consistently performs well and ranks second. This demonstrates its strong performance in enhancing model training for robustness. For instance, let's compare the results for BigCloneBench, where the model trained with *Manifold-Mixup* exhibits a mean value of a maximum probability of 0.9385479. In contrast, the model trained with DropEdge achieves a slightly higher value of 0.9398362. Although DropEdge wins in this case, it's important to note that *Manifold-Mixup* is a close second and still performs well regarding model confidence. In summary, the results in this table show that *Manifold-Mixup* consistently enhances model confidence on test data, which is a valuable characteristic for adversarial robustness.

The results of the robustness experiments with *Manifold-Mixup* align with the outcomes observed with *Mixup*. *Manifold-Mixup*, a variant of *Mixup*, employs a similar mechanism of generating new data by linearly combining two original data points. This data augmentation technique effectively smoothes the data distribution in the feature space. This smoothing effect is critical in reducing the sharpness of decision boundaries between different classes. Adversarial attacks involve making subtle changes to input data in an attempt to mislead the model's predictions. The smoothing effect induced by techniques like *Manifold-Mixup* can make it more difficult for these perturbations to disrupt the model's behavior. Additionally, prior research [101], as cited, has shown that minimizing the loss introduced by *Mixup* approximates minimizing an upper bound on the adversarial loss. *Manifold-Mixup* is a data-adaptive regularization technique that improves robustness.

4.3.3 RQ3: How do existing graph data augmentation methods affect computational resources compared to code data augmentation methods?

Study Design. In this research question, we aim to investigate whether graph data augmentation methods impact the convergence speed of GNN training in source code learning. Faster convergence can lead to more efficient utilization of computational resources [32]. We do this by analyzing the training logs of the models from our previous study in RQ1 and comparing the performance of four graph data augmentation meth-



(a) Java250-GCN (b) Python800-GCN (c) Refactory-GCN

(d) CodRep1-GCN (e) GCJ-GCN (f) BigCloneBench-GCN

Figure 4.6: Training log of GCN in Java250, Python800, Refactory CodRep1, GCJ, and BigCloneBench.

ods against the baseline approaches. Additionally, we aim to gain a deeper understanding of how these data augmentation methods influence the computational resources, which we further explore by repeating the experiments related to the *confidence* of trained source code models as discussed in RQ2. By assessing the convergence speed and its relationship with computational resources, we can evaluate the effectiveness of graph data augmentation methods and their efficiency in practical applications.

Results Analysis. We investigate the convergence speed of the GCN model, as GCN is considered one of the foundational variants of Graph Neural Networks and is widely used in various graph-related applications due to its simplicity and effectiveness [43]. Figure 4.6 displays the training logs of the GCN model on six different datasets. Our empirical findings reveal the following insights:

1) The GCN model's convergence speed appears consistent regardless of the data augmentation methods employed. For example, in cases like Java250-GCN and CodRep1-GCN, the models show limited improvement in accuracy even after

40 epochs. This suggests that the computational resources required for training GNN models with different data augmentation techniques are relatively similar.

2) The results are similar to the findings from the accuracy analysis, with *Manifold-Mixup* consistently outperforming other methods. This highlights the practicality of *Manifold-Mixup*for GNN models, especially in scenarios with limited computational resources.
3) *Manifold-Mixup* might require more training epochs to reach convergence compared to other data augmentation methods. This observation aligns with previous findings [99] that data augmentation methods introducing complexity to the training data often demand more epochs for convergence.
4) In contrast, when compared to *Manifold-Mixup*, code data augmentation methods (i.e., Refactor), as well as dropout-based graph data augmentation methods, show faster convergence. For example, in the case of Refactory-GCN and CodRep1-GCN, data augmentation methods do not significantly improve accuracy, except for *Manifold-Mixup*, even after 60 epochs. This finding suggests that data augmentation methods involving simple program transformations require fewer training epochs to achieve convergence.

We assessed the confidence of trained source code models using different data augmentation methods to investigate how data augmentation methods influence convergence speed. Specifically, we initially calculated the mean values of maximum probabilities of models trained with data augmentation at the 10^{th} training epoch (as reported in Table 4.4). Then, we calculated the difference (as reported in Table 4.7) by subtracting the mean values of maximum probabilities obtained at the 10^{th} training epoch from the values of maximum possibilities achieved after the model is well-trained (as reported in Table 4.3). This approach allows us to understand how data augmentation methods affect the convergence speed of the models.

The results from Table 4.4 suggest that *Manifold-Mixup* might not perform optimally during the early stages of source code model training (e.g., it was the best in only one case). This could be due to the high diversity of data it generates. Linearly mixing two features means synthesizing new and more diverse code data compared to other simple program transformation methods like code refactoring. This observation is particularly noticeable in the case of GCJ-GCN-Refactor.

As training progresses, the performance of *Manifold-Mixup* significantly improves, as evident from the substantial results presented in Table 4.3. This observation indi-

Table 4.7: A case study of mean values of maximum probabilities (the difference value between 10^{th} epoch and well-trained). The best and second-best results are highlighted in gray and blue, respectively.

Model	DA method	GCJ	Refactory	Java250	BigCloneBench
GCN	No Aug	0.0044135	0.2485455	0.0847620	0.3622291
	Refactor	0.0045601	0.2533610	0.0900281	0.3702234
	DropNode	0.0054207	0.2645589	0.0768556	0.3792812
	DropEdge	0.0035604	0.2199798	0.0720799	0.3807038
	Subgraph	0.0016548	0.2531514	0.1133837	0.3699157
	Manifold-Mixup	0.0069290	0.2667552	0.0844159	0.3806153

cates that during the mid-training phase, the model's learning converges rapidly, substantiated by the findings in Table 4.7. In the case of GCN-GCJ, as illustrated in Figure 4.6(e), the green line representing Manifold-Mixup exhibits a more accelerated convergence. This observation is further reinforced by the results in Table 4.7, where Manifold-Mixup achieves the best value of 0.0069290, highlighted in gray.

4.4 Discussion

4.4.1 Limitations

Our empirical study shows that most graph data augmentation methods can significantly contribute to producing more accurate and robust source code models. Therefore, incorporating data augmentation is crucial when tackling source code learning tasks. Thus, incorporating data augmentation is vital when addressing source code learning tasks. Specifically, when employing a suitable data augmentation method, such as *Manifold-Mixup*, the trained source code models exhibit higher accuracy (up to 1.60%) and improved robustness (up to 4.09%) compared to models trained without data augmentation.

Additionally, our results highlight that data augmentation methods specifically designed for source code have limitations in enhancing the performance of source code models.

- Indeed, our observations highlight that no single data augmentation method consistently outperforms others across all datasets. This variability in performance

may be attributed to the influence of data augmentation methods on the underlying data distribution within distinct training datasets. A more in-depth exploration and understanding of this phenomenon could offer an intriguing avenue for future research. For instance, further investigation into the interactions between specific data augmentation strategies and the characteristics of diverse code data could shed light on the factors that potentially affect the effectiveness of these methods.

- The results obtained from our empirical study are focused explicitly on classical classification tasks, encompassing both multi-classification tasks such as authorship attribution and binary classification tasks like bug detection. While these findings offer valuable insights into the effectiveness of data augmentation for these specific tasks, it is essential to note that the applicability of these results to other source code generation tasks, such as program repair and code completion, remains an area to be further explored. Adapting and extending our approach to diverse source code learning scenarios could provide a more comprehensive understanding of the impact of data augmentation on different aspects of code analysis and generation.

4.4.2 Threats to Validity

The study's internal validity is influenced by the implementation of standard GNN training and the selected graph data augmentation methods. The source code for GNN training in various tasks is collected from established sources like Project CodeNet [62], existing research projects [95], and datasets from previous studies [10, 36, 107]. The graph data augmentation methods used, including *Subgraph*, *DropNode*, and *DropEdge*, are applied as per their original implementations. Code refactoring methods for Java and Python are derived from open-source projects on GitHub [18, 66, 83]. This approach maintains the internal validity of the study.

External threats to validity stem from the selection of code-related tasks, datasets, GNN models, and data augmentation methods. The study has carefully chosen tasks and datasets that represent a diverse set of challenges in the field of source code learning. Including various programming languages, such as Java and Python, and different GNN models with distinct architectures and learning mechanisms helps ensure external validity. Additionally, using both code refactoring methods and graph data augmen-

tation methods, selected based on their reputation in the literature, adds to the study's external validity.

Construct threats to validity centered around parameters, randomness, and evaluation measures. The selection of parameters, such as the mixing weight (λ) for *Manifold-Mixup*, adheres to recommendations from previous research, ensuring consistency. The parameters for graph data augmentation methods follow the original releases. To reduce the impact of randomness, experiments are conducted five times, and results are reported as averages with standard deviations, ensuring robust and comprehensive evaluations. These measures help maintain the study's construct validity.

4.5 Conclusion

This paper presents a comprehensive study on the impact of graph data augmentation methods on source code learning, focusing on improving GNN model performance in code-related tasks. The study evaluates widely-used graph augmentation methods, including DropNode, DropEdge, Subgraph, and *Manifold-Mixup*. The research covers four code-related tasks, namely problem classification, bug detection, authorship attribution, and clone detection, using a minimum of four different GNN models for each task. The experimental results indicate that graph data augmentation methods, particularly *Manifold-Mixup*, outperform code data augmentation methods in terms of accuracy and robustness. *Manifold-Mixup*, in particular, is recommended when computational resources are limited, as it enhances the model's ability to learn general features and decision boundaries, leading to improved generalization to unseen graph data. This capability is vital for ensuring a model's performance on training data and real-world graph data. The paper provides valuable insights into the benefits of graph data augmentation in the context of source code learning, offering practical recommendations for researchers and practitioners in the field.

Chapter 5

MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation

5.1 Introduction

Deep learning (DL) has made significant strides in various application domains, including language translation [12], video games [77], and autonomous driving [77]. It has provided remarkable performance in these areas. More recently, researchers in the field of software engineering have started exploring the application of deep learning techniques to automate various code-related tasks. These tasks include code search [24], problem classification [62], and bug detection [67]. Studies [19, 52] in this domain have shown that deep learning, such as *CodeBERT* and *GraphCode2Vec*, can bring substantial benefits to the analysis of source code.

In the realm of DL systems, the performance of deep neural networks (DNNs) is significantly influenced by two key factors: the *model architecture* and the *training data*. Regarding model architecture, when applied to code analysis, it's common practice to adapt natural language processing (NLP) models for source code. For example, Feng *et al.* [19] created *CodeBERT* by modifying *BERT* to handle various downstream code-related tasks effectively. However, obtaining high-quality labeled source code data regarding training data is challenging. The main hurdle is that data labeling requires substantial human effort and domain-specific knowledge. For instance, as men-

tioned in a study [108], labeling code from just a few libraries can take hundreds of person-hours. In summary, data preparation is an indispensable but challenging aspect of developing effective models for code analysis, and this paper specifically emphasizes addressing this crucial issue.

Data augmentation is a technique that addresses the data labeling challenge by generating additional training data through modifications of existing data rather than relying solely on human labeling efforts. The new data samples produced through augmentation are typically semantically consistent with the original data, meaning they retain the same functionalities and labels. This enables the model to learn more effectively and generalize better than using only the original training data. In computer vision and NLP fields, data augmentation has been extensively utilized and studied for training models [17, 68, 106]. For instance, in computer vision tasks, various image transformation techniques like rotation and shear are designed to simulate diverse real-world scenarios that the model may encounter post-deployment. In traditional NLP tasks, a common form of augmentation involves synonym substitution. It is also valuable for exposing the model to a broader range of contexts it might encounter in real-world applications.

Indeed, while data augmentation has demonstrated effectiveness in fields like computer vision and NLP, its application in code analysis is still in its early stages. Researchers in this domain have drawn inspiration from other areas and proposed several data augmentation techniques for code analysis [2, 8, 61, 79, 96]. These techniques often generate additional data through methods like code refactoring or adversarial data generation. However, existing research[16, 97] has indicated that these simple strategies may have limited effectiveness. For instance, studies [6] have shown that merely adding adversarial data to the training set does not significantly improve the generalization of DNN models for code analysis. This raises the open problem of designing data augmentation approaches that can truly enhance the training of DNN models for code analysis.

This paper introduces a novel data augmentation framework called MixCode for source code classification tasks. MixCode is developed to enhance the training process of DNN models. In a broad sense, MixCode follows a similar paradigm to *Mixup* [99], a well-known data augmentation technique initially created for image classification. Mixup linearly combines training data, including their labels, to increase the volume of training data. In the context of source code, MixCode involves two primary steps:

firstly, it generates modified data using various code refactoring techniques, and subsequently, it linearly combines the original code with the transformed code to create new data samples. Our experiments are aimed at assessing the impact and effectiveness of MixCode. These evaluations cover two popular programming languages, Java and Python, and two well-researched code-learning tasks: problem classification and bug detection. Additionally, seven distinct model architectures are considered in the experiments. The implementation of MixCode is available online [4]. Based on that, we answer three research questions as follows:

- RQ1: How effective is MixCode for enhancing the accuracy and robustness of DNNs?
- RQ2: How do different Mixup strategies affect the effectiveness of MixCode?
- RQ3: How does the refactoring method affect the effectiveness of MixCode?

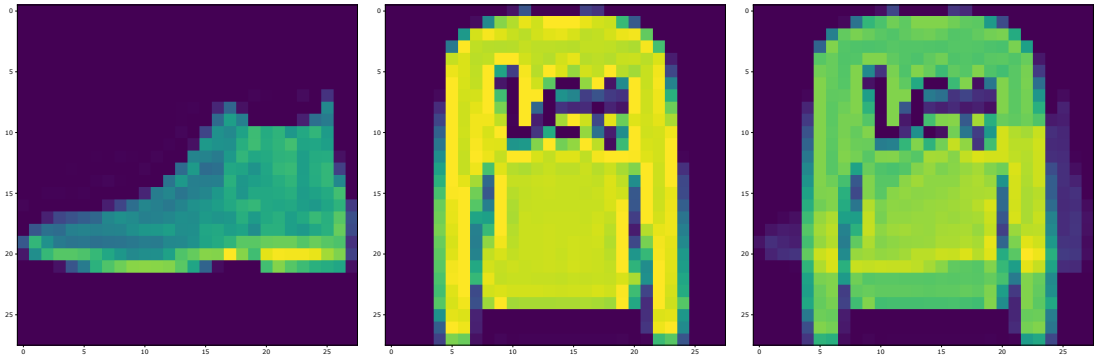
5.2 Preliminaries

The DNN training process involves finding the optimal parameters, such as weights and biases, that allow the model to learn from a given set of training data effectively.

The prepared training data, by its nature, can only capture a limited portion of the entire data distribution. Consequently, the limited volume of training data has become a significant bottleneck that hinders DNN models from achieving peak performance [44]. To address this issue, data augmentation techniques have been proposed to systematically expand the size of the training dataset, thereby enhancing the quality of model training. The fundamental concept behind data augmentation involves generating new data points derived from the existing training data through carefully designed data transformation methods. These transformation techniques aim to modify the data without altering its underlying semantic information. For instance, in image data processing, commonly employed methods include random rotations, padding, and adjustments to brightness [68].

Mixup [99] is a highly effective data augmentation technique initially designed for image classification tasks. Mixup operates through a two-step process: firstly, it randomly selects two data samples from the training dataset, and then, it blends both the data features and the labels of the chosen samples to create a new example for

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



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

Figure 5.1: An example of Mixup for image data. The mixed image is calculated using Eq. (5.1). $\lambda = 0.2$.

training. While its original use case was in image classification, recent research has demonstrated the effectiveness of Mixup in text classification [26]. The technique can be formally represented as follows: given a pair of samples (x^i, y^i) and (x^j, y^j), where x denotes the input features, and y is represented using one-hot encoding to indicate the output label, Mixup generates new data pairs $(x^{ij}_{mix}, y^{ij}_{mix})$:

$$\begin{aligned} x^{ij}_{mix} &= \lambda x^i + (1-\lambda)x^j \\ y^{ij}_{mix} &= \lambda y^i + (1-\lambda)y^j \end{aligned} \tag{5.1}$$

where λ serves as a mixing policy for the input sample pair and is drawn from a Beta distribution characterized by a shape parameter α ($\lambda \sim \text{Beta}(\alpha,\alpha)$). Figure 5.1 provides a visual example of an image generated through Mixup. This technique blends two images into one, allowing the model to extract information from both images.

5.3 MixCode—The Proposed Approach

5.3.1 Overview

Figure 5.2 provides an overview of MixCode in a single epoch. This process involves three main phases:

MixCode begins by loading the raw data from the original code from the initial training set (X, Y). Subsequently, it randomly selects a refactoring method for each

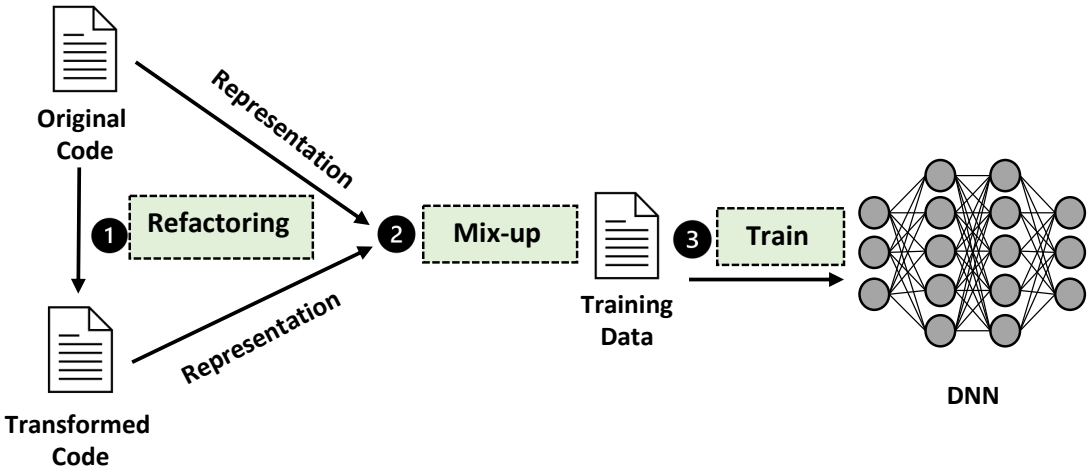


Figure 5.2: Workflow of MixCode within one training epoch.

data point and applies it to the original code, generating the *transformed code*. Code refactoring is a technique that restructures code without altering its semantic behavior [40, 45]. The current version of MixCode supports 18 distinct refactoring methods, which will be explained in more detail later. Consequently, each training epoch involves various sets of transformed code created by different refactoring techniques.

MixCode randomly selects one data point from the original code and another from the transformed code. It then combines these two data points following Eq. (5.1). In this equation, v_x and v_{ref} correspond to x^i and x^j and represent data in the appropriate code format. Typically, the data is in the form of a sequence of token values. Additionally, y_s and y, corresponding to y^i and y^j, are the one-hot encoded label values consistent with the original Mixup approach. This process generates new data pairs (x_{mix}, y_{mix}) like the original Mixup technique and incorporates them into the training set (X_{mix}, Y_{mix}).

The mixed dataset (X_{mix}, Y_{mix}) is used as the training data for that epoch. It's worth noting that while Mixup was initially evaluated on image data in [99], its applicability is not limited to continuous representation spaces. The central idea of Mixup is to combine sample-target pairs to augment the training data size, and it applies to discrete representation spaces for NLP tasks [26]. Similarly, MixCode can be applied to both Integer-type and Float-type input data by setting the input type of embedding layers to float (e.g., `x = torch.tensor(dataset, dtype=torch.float)`).

Table 5.1: Description and examples of 18 types of refactoring methods.

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

Indeed, the performance of MixCode can be influenced by two key factors: the candidate data pairs used for Mixup and the choice of the hyperparameter α. The hyperparameter α plays a crucial role in controlling the percentage (λ) of code content used from both parts for Mixup. Typically, λ is a value within the [0, 1] range sampled from a Beta distribution [99] parameterized by α. The recommended setting for α in the original Mixup work for image classification tasks is 0.2. However, our evaluation explores the impact of different settings for α, ranging from 0.05 to 0.5, to identify an optimal setting for source code classification tasks.

5.3.2 Code Refactoring

Code refactoring is a software development technique to restructure code to improve readability and reduce code complexity while preserving its semantic behavior [40, 45]. The primary goal of code refactoring is to make code more understandable, which helps reduce technical debt. Technical debt refers to the extra work that arises when

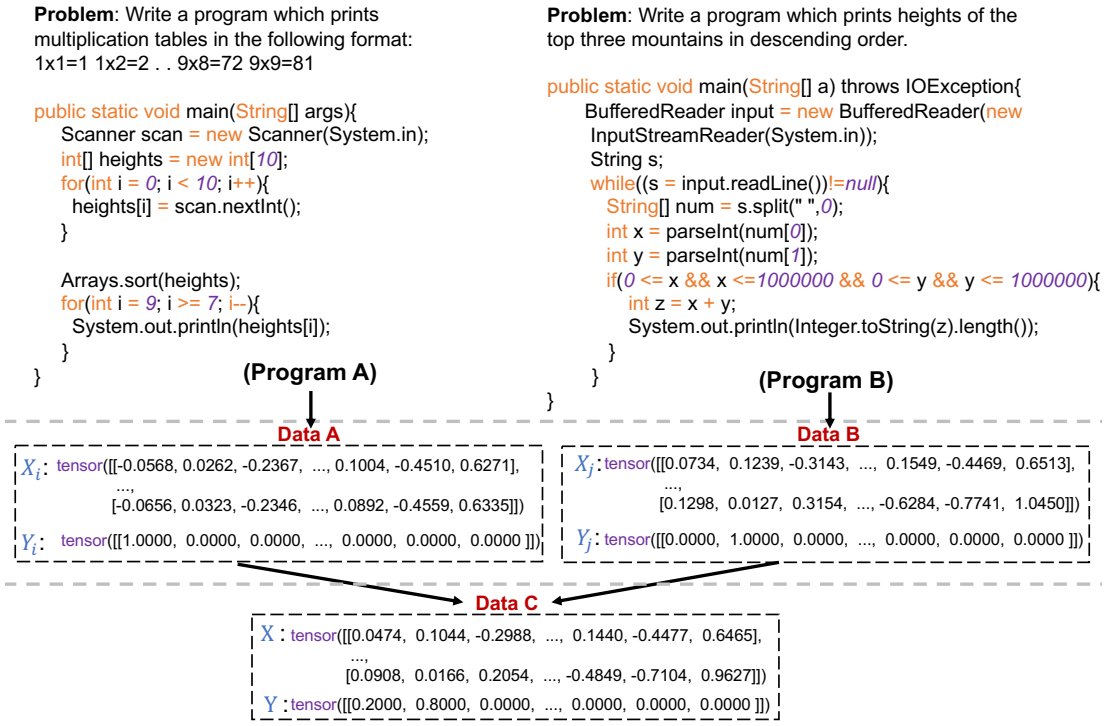


Figure 5.3: An example of two programs mixed by MixCode.

code is poorly structured or not adequately documented. Refactoring enhances code maintainability, making it easier for developers to work with and add new features to the codebase. This is especially important during software maintenance and development processes. Refactoring typically involves making small, standardized modifications to the code. Some common refactoring methods include replacing variables and modifying code, which includes adding or simplifying conditional expressions and method calls. Move features between objects and organize data structures.

MixCode leverages multiple code refactoring methods to generate diverse candidate training data in its first step. It supports 18 refactoring methods listed in Table 5.1 sourced from the literature [61, 86]. These methods aim to enhance the diversity of the data and are designed to be label-preserving, meaning they don't change the code in ways that would affect model decisions. Existing code learning models primarily use two code representations: token sequences and abstract syntax trees (ASTs) [46]. Most of the refactoring methods MixCode supports can be applied to both preprocessing formats, making MixCode highly flexible. These refactoring methods, among others, are

used to create diverse training data for MixCode.

5.3.3 A Case Study

The workflow of MixCode is illustrated in Figure 5.3, which uses an example to demonstrate the process. Here's a step-by-step breakdown: Start with two pieces of source code (Program A and B from the JAVA250 dataset), labeled 0 and 1, respectively. Convert these source code snippets into vectors using CodeBERT [19]. Represent the labels as one-hot vectors. In this case, there are 250 possible classes (as seen in Data A and Data B in Figure 5.3). Linearly mix the code vectors of these two programs using a mixing ratio, λ, set to 0.2. Similarly, linearly mix the label vectors. The mixed code and label vectors (Data C in Figure 5.3) serve as the final input for training the model. This example demonstrates how MixCode combines two source code snippets and their corresponding labels, using a mixing parameter λ in Eq. (5.1), to create a new training data point. The mixing technique is adapted from the original Mixup method for image data and applied to source code in MixCode to augment the training data and enhance model training.

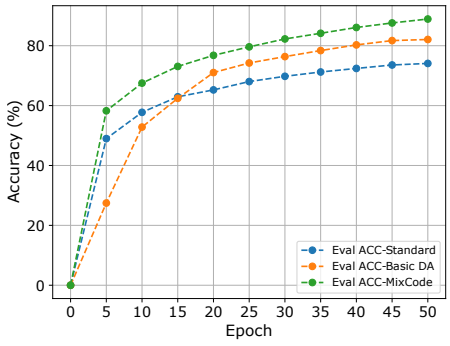


Figure 5.4: Test accuracy of models trained by using different methods (dataset: Java250, model: BagofToken).

Figure 5.4 provides an example of the training process using different methods, highlighting the advantages of MixCode. This example demonstrates that MixCode can improve the model training process, resulting in faster convergence than standard training and traditional data augmentation methods. It aligns with the findings from other fields where Mixup and its variants have successfully enhanced training [30, 31, 101].

5.4 Evaluation

Table 5.2 provides an overview of the datasets and models used to evaluate MixCode. This setup can comprehensively assess the effectiveness of MixCode across different

Table 5.2: 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.

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

programming languages, downstream tasks, and DNN model architectures. This diversity in our experiment design helps ensure a thorough evaluation of the MixCode technique's performance.

In our comparison, we evaluate MixCode against two baseline methods. The first baseline employs a basic data augmentation technique, where the model is trained with random code refactoring applied to the training data at each training epoch. This data augmentation method has previously been used in works like [61] and [96]. The second baseline corresponds to the standard training process, with no data augmentation applied. In our model testing, we assess the performance of DNNs based on test accuracy and robustness. The test accuracy is the percentage of correctly classified data within the entire test dataset. To evaluate the robustness of the models, we generate new test sets by applying code refactoring methods to the original test set. We then calculate the percentage of correctly classified data from this new set. Robustness assesses the model's generalization ability, specifically how well it performs when encountering more diverse, unseen data.

The core code of MixCode is implemented in pure Python, utilizing only the Numpy package. This design allows MixCode to integrate easily into any deep-learning framework for various classification tasks. We've provided two versions of code refactoring methods supporting both Java and Python. We've built the models for the classification tasks using TensorFlow 2.3 and Keras 2.4.3 frameworks. PyTorch 1.6.0 was

Table 5.3: Effectiveness of MixCode concerning test accuracy and robustness (average± standard deviation, %) on original test data and transformed test data. Standard: standard training without data augmentation. Basic: basic data augmentation. The value with a gray background indicates the best result, and the value highlighted in red shows the accuracy difference between the best and the second-best. The higher, the better.

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

used to construct the models for the bug detection tasks. We configured the training epochs to be 50 for SeqOfToken, BagOfToken, CodeBERT, and GraphCodeBERT experiments and 100 for the GNN experiments. Our experiments were conducted on an NVIDIA Tesla V100 16G SXM2 GPU. To ensure the reliability of our results, each model was trained five times, and we reported the average results along with the standard deviation.

5.4.1 RQ1: Effectiveness of MixCode

The results in the left component of Table 5.3 illustrate the test accuracy of the trained models on the original test data. One key observation is that MixCode consistently outperforms the two baselines across different datasets and models. This indicates that data augmentation significantly enhances model performance compared to standard training. It's worth noting that simply adding data to the training set, as done in the primary data augmentation, leads to only minor improvements, especially in bug detection tasks (CodRep1 and Refactory). For instance, in Refactory with GAT, essential data augmentation improves accuracy by a maximum of 0.29%. This aligns with findings from prior work [96], which suggested that traditional adversarial training (simply

adding adversarial examples to the training data) cannot boost the robustness of source code models. In contrast, MixCode shows a substantial improvement, with gains of up to 5.69% compared to primary data augmentation. In bug detection tasks (CodRep1 and Refactory), where primary data augmentation is ineffective, MixCode achieves remarkable accuracy improvements, with gains of up to 6.24%

The results in the right part of Table 5.3 illustrate the robustness of various trained models when tested on the transformed test data. It's important to note that the transition from test accuracy to robustness reveals a significant decline in the performance of all standard-trained models. For example, the accuracy of the JAVA250-BagofToken model drops from 71% to 40%. This decline underscores the poor generalization ability of models trained solely on original data; they struggle to handle unseen data, even if it has the same functionality as the training data. However, the drop in performance is mitigated when data augmentation is employed, especially when using MixCode. MixCode consistently achieves the best results in all cases, significantly improving robustness. In particular, it outperforms essential data augmentation by up to 26.06%, with an average improvement of 5.58%, a remarkable achievement. Notably, in the problem classification task (JAVA250 and Python800), all models experience more than a 20% improvement in accuracy. Perhaps surprisingly, the robustness is close to the original test accuracy, such as in the case of Python800-BagofToken, where the original accuracy is 68.27% and the robustness is 63.65%. In the bug detection task (CodRep1 and Refactory), although the improvement is less pronounced than in problem classification (JAVA250 and Python800), it is still quite promising, ranging from 2.87% to 5.47%. This highlights the effectiveness of MixCode in enhancing the model's robustness against transformed data.

5.4.2 RQ2: Impact of Mixup Strategies

In the default configuration of MixCode, a pair of randomly selected original and transformed codes are mixed (Ori+Ref). To understand the effectiveness of this mixing strategy, we compared (Ori+Ref) with two other Mixup strategies: original code mixing original code (Ori+Ori) and transformed code mixing transformed code (Ref+Ref). This analysis aims to determine which strategy is the most beneficial for data augmentation and model improvement.

Table 5.4 illustrates the comparative effectiveness of the three Mixup strategies:

Table 5.4: 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			Dataset	DNN	Mixup Strategy		
		Ori+Ori	Ori+Ref	Ref+Ref			Ori+Ori	Ori+Ref	Ref+Ref
		Test Accuracy					**Robustness**		
JAVA250	**BagofToken**	73.09±0.03	82.32±0.07	82.13±0.03	**JAVA250**	**BagofToken**	43.87±0.03	78.17±0.05	66.62±0.03
	SeqofToken	94.49±0.08	94.52±0.23	93.18±0.15		**SeqofToken**	78.27±0.02	85.60±0.01	84.30±0.02
	CodeBERT	96.54±0.02	96.98±0.09	95.17±0.16		**CodeBERT**	91.17±0.22	93.17±0.11	93.01±0.13
	GraphCodeBERT	96.76±0.04	97.16±0.12	95.56±0.19		**GraphCodeBERT**	92.01±0.19	94.07±0.16	93.56±0.17
CodRep1	**GGNN**	64.37±0.25	65.42±0.35	65.38±0.33	**CodRep1**	**GGNN**	42.40±0.02	55.01±0.01	49.53±0.02
	GCN	63.42±0.26	64.18±0.44	63.89±0.27		**GCN**	42.08±0.02	54.03±0.03	49.42±0.03
	GAT	63.29±0.47	64.09±0.36	63.78±0.35		**GAT**	41.66±0.02	53.81±0.03	48.78±0.02
	CodeBERT	72.77±0.07	72.96±0.05	71.14±0.03		**CodeBERT**	63.26±0.06	66.15±0.09	65.95±0.03
	GraphCodeBERT	73.21±0.11	73.34±0.13	71.78±0.16		**GraphCodeBERT**	63.17±0.08	65.91±0.13	64.76±0.16
Python800	**BagofToken**	68.27±0.08	67.88±0.07	65.67±0.09	**Python800**	**BagofToken**	40.46±0.01	63.65±0.02	62.35±0.03
	SeqofToken	84.68±0.04	85.00±0.20	81.22±0.45		**SeqofToken**	75.15±0.02	82.94±0.02	78.54±0.01
	CodeBERT	96.35±0.04	96.79±0.06	95.11±0.18		**CodeBERT**	90.08±0.05	92.16±0.08	92.08±0.03
	GraphCodeBERT	96.87±0.05	97.09±0.08	95.16±0.21		**GraphCodeBERT**	91.87±0.08	94.56±0.07	93.12±0.09
Refactory	**GGNN**	82.46±0.28	88.22±0.18	86.58±0.22	**Refactory**	**GGNN**	68.09±0.03	72.81±0.03	69.69±0.02
	GCN	81.71±0.24	84.12±0.17	84.61±0.25		**GCN**	65.68±0.02	67.73±0.03	66.36±0.01
	GAT	80.22±0.27	82.68±0.12	83.38±0.31		**GAT**	62.37±0.02	66.49±0.01	65.09±0.02
	CodeBERT	96.98±0.41	97.57±0.37	95.39±0.39		**CodeBERT**	92.67±0.06	95.41±0.06	93.44±0.07
	GraphCodeBERT	97.78±0.27	98.16±0.21	96.03±0.19		**GraphCodeBERT**	92.45±0.09	95.23±0.11	93.17±0.13

Ori+Ref (original code mixed with transformed code), Ori+Ori (original code mixed with original code), and Ref+Ref (transformed code mixed with transformed code). Here, we assessed their impact on model performance across various tasks. First, in comparison to standard training results from Table 5.3, it's evident that all three Mixup strategies provide a performance boost. In most cases (84 out of 108), Mixup strategies outperform essential data augmentation. Next, when comparing the different Mixup strategies, the results indicate that, for Java language tasks (JAVA250, CodRep1), Ori+Ref consistently yields the best results across all tasks regarding test accuracy. Ref+Ref, the second-best strategy, is slightly better than Ori+Ori, but overall, the differences between these three strategies are minimal, e.g., 94.49% vs 94.52%, 93.18%. No single strategy consistently outperforms the others for Python language tasks (Python800, Refactory). However, Ori+Ref typically ranks in the top two positions in all cases. Even when it's not the best, the performance gap between Ori+Ref and the best strategy is minor (e.g., 84.12% vs 84.61%). Unlike Java tasks, the distinctions between these three combinations become more significant in Python tasks, e.g., 82.46% vs 88.22% vs 86.58%. Consequently, if test accuracy is the primary consideration, Ori+Ref is recommended as the MixCode combination. Regarding robustness, Ori+Ref consistently and significantly outperforms the other two combinations, with

Table 5.5: Results of MixCode using Ori+Ref strategy for Mixup with different α. The value with a gray background indicates the best result. The higher, the better.

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

an average improvement in robustness of 2.6% compared to the second-best combination. These results suggest that using original and transformed data is a superior choice for MixCode when training robust models.

In our study, we investigated how the hyperparameter α in the Beta distribution, which determines the value of λ in Eq. (5.1), affects the performance of MixCode. The α parameter can be set across a range of values, and we explored the range from 0.05 to 0.5, consistent with the recommended setting of $\alpha = 0.2$ as per the original Mixup technique. Table 5.5 presents the test accuracy and robustness results for models trained with different values of α. It's important to note that when α is set to 0.5, the models based on SeqofToken consistently perform poorly, with less than 1% accuracy. This phenomenon suggests that when α and λ become larger, the mixed code may become meaningless, leading to models that can't effectively learn from the data. We've identified this as an intriguing aspect for future research. However, from the remaining results, it's evident that MixCode tends to outperform standard training and essential data augmentation across most scenarios, regardless of the specific value of α. This demonstrates that a smaller value of α often produces better models. In 13 out of 18 cases and 15 out of 18 cases, α values of 0.05 or 0.1 yielded models with higher accuracy and better robustness. These findings underline the importance of selecting an appropriate α value for specific tasks and datasets to achieve optimal results with MixCode.

Table 5.6: Refactoring methods selection (test accuracy). Good/poor: MixCode using the best/worst nine refactoring methods. Baseline: MixCode using all 18 methods. The best method of 18 for each Dataset is highlighted with a gray background.

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

5.4.3 RQ3: Impact of Refactoring Methods

We delve into the impact of the specific refactoring methods on MixCode's performance, mainly focusing on the best Mixup strategy *Ori+Ref* identified as the top-performing strategy in RQ1 and RQ2. The aim is to rank the refactoring methods based on their influence on the trained model's performance. Firstly, we train the model independently using each refactoring method. This allows us to rank the refactoring techniques based on their impact on the model's performance. Based on the ranking, we divide the 18 refactoring methods into two sets: "good" (high-ranking methods) and "poor" (low-ranking methods). The intention is to evaluate how the quality of refactoring methods impacts MixCode's effectiveness. We then apply MixCode using these two sets of refactoring methods separately. The goal is to determine if using a set

Table 5.7: Refactoring methods selection (robustness). Good/poor: MixCode using the best/worst nine refactoring methods. Baseline: MixCode using all 18 methods. The best method of 18 for each Dataset is highlighted with a gray background.

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

of better refactoring methods can further enhance MixCode's performance. This approach systematically explores whether MixCode can be optimized by selecting more effective refactoring methods. Our focus on two specific models, BagofToken for problem classification and GGNN for bug detection, allows for a more in-depth analysis of MixCode's improvements, as it's often easier to observe differences in performance with specific models.

The results in Table 5.6 present the test accuracy of models trained using individual program refactoring methods on the original test data. Surprisingly, MixCode can yield more accurate models when employing a single refactoring method. The difference can be as substantial as 4.92%, as seen when comparing *Arguments Adding* to the *Baseline*in the Java dataset. To identify the most effective refactoring methods,

we set a threshold of accuracy greater than 86.00%, 65.75%, 68.00%, and 89.00% for the JAVA250, CodRep1, Python800, and Refactory datasets, respectively, categorizing them as ”Good.” The remaining methods are classified as ”Poor.” Notably, when focusing solely on test accuracy, MixCode using the ”Good” refactoring methods outperforms the ”Poor” methods. On average, ”Good” methods exhibit a 1.98% improvement in test accuracy compared to ”Poor” ones. These results demonstrate that MixCode can effectively utilize a single well-selected refactoring method if the primary goal is to improve the test accuracy of the trained model. Furthermore, combining superior refactoring methods with higher single-test accuracy can result in a more effective MixCode.

Table 5.7 presents the results of trained models concerning robustness. Initially, compared with standard training (in Table 2), MixCode employing a single refactoring method consistently generates more robust models, with only one exception: ”Local Variable Adding - Refactory” exhibits lower robustness than standard training. Subsequently, we compare models trained with a single refactoring method to those trained with a combination of multiple methods. In contrast to the results based solely on test accuracy, the findings demonstrate that MixCode using a single refactoring method cannot surpass the performance of MixCode with multiple methods. The difference is substantial, as evidenced by the example in the JAVA250 dataset. While the robustness of single-method models ranges from 42.42% to 68.34%, the robustness of using the ”Poor” refactoring methods can reach 78.19%, resulting in an approximate 10% difference in robustness. This discrepancy is reasonable since compelling the model to focus solely on learning a specific code refactoring might limit the model’s generalization ability. Additionally, we compare the performance of ”Good” and ”Poor” methods. Surprisingly, in half of the cases (2 out of 4), ”Poor” methods exhibit higher robustness than ”Good” methods. This finding indicates that models trained with better refactoring methods (possessing higher original test accuracy) don’t consistently outperform others in terms of robustness. It highlights the existence of a trade-off between original test accuracy and robustness when selecting refactoring methods for MixCode.

5.5 Discussion

5.5.1 Limitations

We noted that the choice of hyperparameter α can significantly impact the performance of MixCode. In the worst-case scenario, such as when $\alpha = 0.5$, certain models might struggle to extract valuable insights from the mixed data, raising legitimate concerns about the utility of MixCode. Nevertheless, it is conceivable to overcome this limitation by employing MixCode with a smaller α value. Prior research in the field of source code analysis predominantly emphasizes the introduction of novel code representation and code embedding techniques, as demonstrated in studies such as [25, 52]. Alternatively, some research delves into leveraging large pre-trained language models to enhance downstream code-related tasks [56]. Yet, relatively few studies have devoted their attention to the quality of training data [56] or the optimization of training methodologies. This landscape reveals an array of promising research prospects in the realm of source code analysis, ripe for further investigation.

5.5.2 Threats to Validity

Internal validity concerns stem from implementing standard training, essential data augmentation, and MixCode. The training datasets for the classification tasks (JAVA250, Python800) are sourced from the CodeNet project [62], while the defect detection tasks (CodRep1, Refactory) are drawn from [10, 36, 107] Mixup's implementation is consistent with its original release [99] For the Java language, we leverage 18 refactoring methods from [61, 86], adapting their implementation to the Python language where needed.

External threats to validity relate to the chosen source code tasks, datasets, deep neural networks, and refactoring methods. We consider two diverse tasks, problem classification and bug detection, encompassing two datasets for each task. Our study extends to two of the most widely used programming languages, Java and Python, catering to software developers. We cast a wide net by employing seven distinct types of deep neural networks, including renowned pre-trained programming language models. Furthermore, our selection of refactoring methods incorporates those commonly found in the literature, ensuring a comprehensive evaluation.

Construct threats to validity arise primarily from the parameters of MixCode, ran-

domness, and our choice of evaluation metrics. MixCode mainly hinges on the parameter λ, which regulates the degree of input instance mixing. We align our parameter choices with the recommendations of the original Mixup algorithm and investigate their influence. MixCode consistently outperforms standard training and essential data augmentation regardless of the parameter setting. To mitigate the impact of randomness, we conducted each experiment five times, reporting average and standard deviation results. Our evaluation metrics encompass the original test data's accuracy and the transformed test data's robustness, providing insights into the generalization capabilities of DNNs.

5.6 Conclusion

This paper introduced MixCode, a novel data augmentation framework for source code analysis. MixCode represents a significant advancement by facilitating enhanced model performance without gathering or annotating new code. In essence, MixCode supports an array of 18 refactoring methods and is designed to be extensible to accommodate new techniques. We conducted extensive experiments to assess MixCode's efficacy across a spectrum of parameters and conditions. This comprehensive evaluation included two programming languages (Java and Python), two fundamental code tasks (problem classification and bug detection), two pre-trained models (CodeBERT and GraphCodeBERT), four datasets (JAVA250, Python800, CodRep1, and Refactory), and five diverse deep neural network architectures (BagofToken, SeqofToken, and GAT). Our experimental results unequivocally revealed that MixCode consistently outperforms the essential data augmentation baseline, showcasing accuracy improvements of up to 6.24% and an astonishing 26.06% boost in robustness. Our configuration analysis established optimal MixCode performance when linearly blending the original and transformed code. In the future, our research will delve into integrating an even more comprehensive array of refactoring methods and extending support to various source code tasks.

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

The importance of data quality and diversity through techniques like data augmentation is well-established in fields such as computer vision and natural language processing. However, in source code learning, the existing practice of data augmentation is in its early stages and is primarily limited to simple program transformations. To overcome these challenges and advance the field of data augmentation in source code learning.

In Chapter 3, we initiate an early investigation into the applicability of data augmentation techniques designed for textual data in source code learning. Our focus is on code classification tasks, and we conduct a comprehensive empirical analysis across four critical code-related problems and four different DNN architectures. Our findings not only reveal the data augmentation techniques that improve the accuracy and robustness of models for source code learning but also highlight their effectiveness, even when they introduce minor deviations in the source code syntax.

In Chapter 4, we introduce a novel approach by leveraging data augmentation techniques designed initially for graph-structured data in the field of graph learning. The objective is to investigate whether these innovative data augmentation methods outperform traditional code refactoring methods to produce more accurate and robust models.

In Chapter 5, we introduce a data augmentation approach inspired by *Mixup* in computer vision. The method uses various code refactoring techniques to generate transformed code instances that maintain consistent labels with the original data. We then adapt the Mixup technique to combine these transformed code samples with the

original code, effectively augmenting the training data. Our comprehensive evaluation of this approach includes two programming languages, two code-related tasks, four benchmark datasets, and seven different DNN architectures.

As a result, this thesis fills the gaps in data augmentation for source code learning.

6.2 Future Work

In the future, to continue advancing the field of data augmentation for code model training, we have two main objectives: (1) We plan to investigate how combining with other confidence scores, such as uncertainty metrics and active learning methods, can further improve the quality of data selection. This research will help us refine the data augmentation process and enhance the training of code models. (2) We aim to assess how well source code data augmentation techniques perform when applied to fine-tune open-source large language models like Llama. This analysis will provide insights into the applicability and benefits of data augmentation in refining existing pre-trained models for code understanding. These future research directions will contribute to the continued development and improvement of data augmentation methods for code model training, enhancing the accuracy and robustness of models in source code-related tasks.

Bibliography

[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)*, 51(4):1–37, 2018.

[2] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[4] P. Baldi and P. J. Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, volume 26, 2013.

[5] P. Bielik and M. Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

[6] P. Bielik and M. Vechev. Adversarial robustness for code. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 896–907. PMLR, 13–18 Jul 2020.

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

[8] N. D. Bui, Y. Yu, and L. Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Pro-*

ceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 511–521, 2021.

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

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

[11] N. Chirkova and S. Troshin. Empirical study of transformers for source code. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 703–715, 2021.

[12] 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)*, pages 1723–1732, 2015.

[13] Z. Dong, Q. Hu, Y. Guo, M. Cordy, M. Papadakis, Y. L. Traon, and J. Zhao. Enhancing code classification by mixup-based data augmentation. *arXiv preprint arXiv:2210.03003*, 2022.

[14] Z. Dong, Q. Hu, Y. Guo, M. Cordy, M. Papadakis, Z. Zhang, Y. Traon, and J. Zhao. Mixcode: Enhancing code classification by mixup-based data augmentation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 379–390, mar 2023.

[15] Z. Dong, Q. Hu, Y. Guo, Z. Zhang, M. Cordy, M. Papadakis, Y. L. Traon, and J. Zhao. Boosting source code learning with data augmentation: An empirical study. *arXiv preprint arXiv:2303.06808*, 2023.

[16] Z. Dong, Q. Hu, Z. Zhang, and J. Zhao. On the effectiveness of graph data augmentation for source code learning. *Knowledge-Based Systems*, page 111328, 2023.

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

[18] W. Feng, J. Zhang, Y. Dong, Y. Han, H. Luan, Q. Xu, Q. Yang, E. Kharlamov, and J. Tang. Graph random neural networks for semi-supervised learning on graphs. *Advances in neural information processing systems*, 33:22092–22103, 2020.

[19] 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.

[20] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1147–1158. IEEE, 2020.

[21] S. Geisler, T. Schmidt, H. Şirin, D. Zügner, A. Bojchevski, and S. Günnemann. Robustness of graph neural networks at scale. In *Neural Information Processing Systems, NeurIPS*, 2021.

[22] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. volume 70, pages 1263–1272, 2017.

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

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

[25] 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.

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

[27] 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.

[28] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

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

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

[31] D. Hendrycks, A. Zou, M. Mazeika, L. Tang, B. Li, D. Song, and J. Steinhardt. Pixmix: Dreamlike pictures comprehensively improve safety measures. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16783–16792, 2022.

[32] A. Hernández-García and P. König. Data augmentation instead of explicit regularization. *arXiv preprint arXiv:1806.03852*, 2018.

[33] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[34] Q. Hu, Y. Guo, X. Xie, M. Cordy, L. Ma, M. Papadakis, and Y. L. Traon. Codes: A distribution shift benchmark dataset for source code learning. *arXiv preprint arXiv:2206.05480*, 2022.

[35] 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)*, pages 200–20010. IEEE, 2018.

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

[37] H. Jebnoun, H. Ben Braiek, M. M. Rahman, and F. Khomh. The scent of deep learning code: An empirical study. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 420–430, 2020.

[38] B. Jiang, Y. Chen, B. Wang, H. Xu, and B. Luo. Dropagg: Robust graph neural networks via drop aggregation. *Neural Networks*, 163:65–74, 2023.

[39] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[40] A. Kaur and M. Kaur. Analysis of code refactoring impact on software quality. In *MATEC Web of Conferences*, volume 57, page 02012. EDP Sciences, 2016.

[41] M. Kimura. Why mixup improves the model performance. In *Artificial Neural Networks and Machine Learning – ICANN 2021: 30th International Conference on Artificial Neural Networks, Bratislava, Slovakia, September 14–17, 2021, Proceedings, Part II*, page 275–286, 2021.

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

[43] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

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

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

[46] T. H. M. Le, H. Chen, and M. A. Babar. Deep learning for source code modeling and generation: models, applications, and challenges. *ACM Comput. Surv.*, 53(3), jun 2020.

[47] B. Li, Y. Hou, and W. Che. Data augmentation approaches in natural language processing: A survey. *AI Open*, 3:71–90, 2022.

[48] H. Li, C. Miao, C. Leung, Y. Huang, Y. Huang, H. Zhang, and Y. Wang. Exploring representation-level augmentation for code search. In *EMNLP*, pages 4924–4936, 2022.

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

[50] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

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

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

[53] A. Maćkiewicz and W. Ratajczak. Principal components analysis (pca). *Computers & Geosciences*, 19(3):303–342, 1993.

[54] V. Marivate and T. Sefara. Improving short text classification through global augmentation methods. In *Machine Learning and Knowledge Extraction*, pages 385–399, 2020.

[55] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438*, 2023.

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

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

[58] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo. An empirical comparison of pre-trained models of source code. *arXiv preprint arXiv:2302.04026*, 2023.

[59] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR, 2021.

[60] L. Perez and J. Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[61] 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)*, pages 36–46. IEEE, 2021.

[62] 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.

[63] S.-A. Rebuffi, S. Gowal, D. A. Calian, F. Stimberg, O. Wiles, and T. A. Mann. Data augmentation can improve robustness. *Advances in Neural Information Processing Systems*, 34:29935–29948, 2021.

[64] K. Ren, T. Zheng, Z. Qin, and X. Liu. Adversarial attacks and defenses in deep learning. *Engineering*, 6(3):346–360, 2020.

[65] S. Ren, J. Zhang, L. Li, X. Sun, and J. Zhou. Text AutoAugment: Learning compositional augmentation policy for text classification. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021.

[66] Y. Rong, W. Huang, T. Xu, and J. Huang. Dropedge: towards deep graph convolutional networks on node classification. In *ICLR*, 2020.

[67] 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.

[68] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[69] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu. Learning program semantics with code representations: An empirical study. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 554–565. IEEE, 2022.

[70] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

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

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

[73] 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 *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE, 2014.

[74] D. A. Van Dyk and X.-L. Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001.

[75] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *stat*, 1050:20, 2017.

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

[77] 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*, 575(7782):350–354, 2019.

[78] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066. PMLR, 2013.

[79] 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*, pages 287–298, 2022.

[80] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[81] X. Wang, X. Liu, P. Zhou, Q. Liu, J. Liu, H. Wu, and X. Cui. Test-driven multi-task learning with functionally equivalent code transformation for neural code generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.

[82] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[83] Y. Wang, W. Wang, Y. Liang, Y. Cai, and B. Hooi. Graphcrop: Subgraph cropping for graph classification. *arXiv preprint arXiv:2009.10564*, 2020.

[84] Y. Wang, W. Wang, Y. Liang, Y. Cai, and B. Hooi. Mixup for node and graph classification. In *Proceedings of the Web Conference 2021*, pages 3663–3674, 2021.

[85] 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.

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

[87] 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.

[88] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

[89] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[90] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[91] C. S. Xia and L. Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[92] Q. Xie, Z. Dai, E. Hovy, T. Luong, and Q. Le. Unsupervised data augmentation for consistency training. *Advances in neural information processing systems*, 33:6256–6268, 2020.

[93] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[94] 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.

[95] 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*, pages 1482–1493, 2022.

[96] N. Yefet, U. Alon, and E. Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[97] S. Yu, T. Wang, and J. Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022.

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

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

[100] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

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

[102] 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*, 31, 2018.

[103] 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*, pages 1159–1170, 2020.

[104] 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*, pages 185–188, 2020.

[105] T. Zhao, G. Liu, S. Günnemann, and M. Jiang. Graph data augmentation for graph machine learning: A survey. *arXiv preprint arXiv:2202.08871*, 2022.

[106] 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*, volume 35, pages 11015–11023, 2021.

[107] H. Zhong and Z. Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE, 2015.

[108] 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*, 32, 2019.

Published Papers

1 **Zeming Dong**, Qiang Hu, Yuejun Guo, Maxime Cordy, Mike Papadakis, Zhenya Zhang, Yves Le Traon, and Jianjun Zhao. MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation. In *2023 IEEE 30th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 379-390, March 2023.

2 **Zeming Dong**, Qiang Hu, Yuejun Guo, Zhenya Zhang, and Jianjun Zhao. Boosting Source Code Learning with Text-Oriented Data Augmentation: An Empirical Study. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pp. 383-392, October 2023.

3 **Zeming Dong**, Qiang Hu, Zhenya Zhang, and Jianjun Zhao. On the Effectiveness of Graph Data Augmentation for Source Code Learning. Accepted by *Knowledge-Based Systems*, December 2023.