Data Augmentation for Supervised Code Translation Learning

Anonymous Author(s)

ABSTRACT

Data-driven program translation has been recently the focus of several lines of research. A common and robust strategy is supervised learning. However, there is typically a lack of parallel training data, i.e., pairs of code snippets in source and target language. While many data augmentation techniques exist in the domain of natural language processing, they cannot be easily adapted to tackle code translation due to the unique restrictions of programming languages. In this paper, we develop a novel rule-based augmentation approach tailored for code translation data, and a novel retrieval-based approach that combines code samples from unorganized big code repositories to obtain new training data. Both approaches are language-independent. We perform an extensive empirical evaluation on existing benchmarks showing that our method improves the accuracy of state-of-the-art supervised translation techniques by up to 35%.

ACM Reference Format:
Anonymous Author(s). 2023. Data Augmentation for Supervised Code Translation Learning. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Some programs or software systems require multiple versions in different languages. For example, open-source prototypes in academia are often subject to translation into new programming languages. Researchers who wish to port codebases from obsolete or deprecated languages to modern languages or platforms often need to rewrite these programs [43]. In practice, software development frequently requires program translation. Rewriting software in the required language is both time-consuming and laborious. For example, the Commonwealth Bank of Australia spent around $750 million and five years translating its platform from COBOL to Java [43].

1.1 Code translation methods

The traditional methods of **rule-based** compilers or cross-language interpreters are hardwaired and require extensive human adaptation and are limited to a small set of programming languages [2]. Recently, several automated program translation and code migration approaches emerged [11, 36, 43].

Supervised learning has shown great promise in this regard [11, 36]. This approach requires a parallel dataset, which consist of pairs of code snippets in the original language and their corresponding translation in the target language.

Table 1: Comparison of supervised and unsupervised translation on Java-C# dataset [8, 30]

Genre	Method	Description	Program Accuracy
Supervised Translation	Tree2tree	tree-to-tree neural networks	70.1%
Unsupervised Translation	TransCoder	weakly-supervised neural translation	49.9%
Large Language Models	RoBERTa (code)	Transformer -based	56.1%
	CodeBERT		59.0%

In reality, parallel datasets for arbitrary language pairs are often amiss making it difficult to apply supervised learning. One direction to circumvent this problem is to resort to **weakly-supervised or unsupervised learning** [43, 44]. Facebook AI proposed to pre-train the translation model on the task of denoising randomly corrupted programs and to optimize the model through back translation [43]. Yet, weak supervision does not provide the same level of accuracy as strongly supervised methods. [43]. In the experiment that is shown in Table 1, one sees a rare comparison between supervised and unsupervised methods. Here, the unsupervised method performs worse than the state-of-the-art supervised model Tree2tree that was trained on a parallel dataset of Java to C# translation pairs [8]. This is consistent with the experience from natural language translation, where the supervised models are shown to perform better than unsupervised models, when the labeled training data is sufficiently available [14, 24]. Further most unsupervised or weakly-supervised machine translation methods heavily depend on back-translation [43, 44], a technique initially developed for natural language translation that involves training on noisy inputs. While this method is attainable for natural languages, where minor inaccuracies might not significantly impact the sentence's meaning [44], its noise-tolerance has heavier consequences on code translation. Even a single token alteration can lead to compilation failures or incorrect program outcomes.

Recently, the booming development of **Large Language Models** (LLMs) has brought new research directions to code translation, such as transformer-based models BERT [15] and GPT [7]. These models leverage pre-trained contextual understanding models from a large number of diverse datasets containing code snippets and natural language text. However, the performance of LLMs in real-world code translation is not satisfactory. The experiment results by Lu et al. shown in Table 1 reveal that the performance of RoBERTa (code) and CodeBERT on the same dataset is worse than the supervised method [30]. Pan et al. also evaluated code translation on recent commercial LLM GPT-4 [40]. Results show that the performance lacks reliability and stability with incorrect translations ranging from 52.7% to 97.9%, involving 14 root causes for translation bugs after analysis. According to their experiment, even with adding more natural language comments to the code and refining the prompt, the performance of LLMs can only be improved by 5.5% [40].

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

With all the aforementioned limitations that still exist for unsupervised and weakly-supervised techniques, we deem it worthwhile to further explore the avenue of training data augmentation to support supervised translation techniques.

1.2 Data augmentation for supervised code translation (current state)

Existing training data for code translation is mainly created manually. Data augmentation aims at generating additional data points based on the empirically observed training data without explicitly collecting new data [6, 16, 19, 48]. Our goal is to explore the potential of supervised translation with the help of DA.

The two lines of research that are closely related to augmentation for code translation are **data augmentation for natural language translation** [17, 48], which is not directly applicable to code, and **data augmentation for bug fixing**, which is not directly applicable to the translation task [5, 39].

There have been significant advances towards augmentation of natural language data for translation [37, 45, 48]. However, one cannot directly apply natural language augmentation techniques for code translation. Natural language words are often fungible, allowing us to comprehend the text even if one or more words are replaced. This feature makes the text more manageable to maintain a set of invariants. The commonly used approaches are word substitution [17, 48] and back-translation [37, 45]. Vanilla word substitution can hardly be used for code because of the programming language's strict grammar and open vocabulary. A minor edit may alter program semantics and syntax or cause code failure. Furthermore, the freely and continuously evolving name vocabularies pose another non-trivial challenge for regulating a code augmentation procedure. Therefore, there is a need to find a customized method to augment code data specific for code translation tasks. Furthermore, there is also a lack of mapping dictionaries for programming languages impeding a parallel substitution in a target language.

Only recently code augmentation has been explored for bug fixing [5, 39]. These approaches are based on manual rule generation, which is hard to generalize to more bug cases not to mention the entirely different task of code translation, which requires *parallel* training data, i.e., pieces of code and corresponding translations.

Large Language Models can also be considered for data augmentation in supervised code translation models due to their ability to generate additional code based on specified prompts. However, utilizing machine-generated code from LLMs for training data presents significant challenges. First, a substantial portion of the code generated by LLMs contains API misuses, 62% with the latest models such as GPT-4 [51]. Second, LLM-generated code frequently exhibits issues, such as missing or irrelevant algorithmic steps, and small [38], hard-to-detect bugs that are less frequent in human-written code [21]. Parallel code with such quality issues can severely harm the deep learning process [46, 50]. Finally, existing LLM solutions require natural language descriptions or comments for fine-tuning [21, 40], whereas, for data augmentation, only raw code data may be available, further complicating their suitability for this purpose.

To avoid the limitations of machine generated code, we intend to pursue approaches do not rely on intractable code generation.

1.3 Research question and our methodology

We aim to solve the following **Research question.** Given a supervised code translation model and a corresponding training dataset, we want to understand the efficacy of existing augmentation methods and develop an effective method that is tailored to the task of supervised code translation.

Our methodology. We explored three directions to augment training data for code translation. First, we directly apply rule-based and back-translation techniques from NLP to code. Second, by modifying the rules for bug fixing [5, 39], we design our own **rule-based approach** with rules that are tailored to the task of code translation. In our experiments, we observe that rule-based approaches cannot yield diversification for the training data, and that back-translation is highly dependent on the quality of the initial model (Section 5.1). Thus, we explored a third fundamentally different approach - **retrieval-based approach**. To make sure that the augmented code is real and contains unseen code patterns, we use code retrieval to find additional code and its potential translation in Big Code [8, 9]. This approach circumvents the problem of syntax and semantics tampering caused by artificial code generation. It has already been used in other programming applications such as code translation [8] and code summarization [28]. However, as translation retrieval is on its own susceptible to inaccurate results, which can cause noise, we developed optimization functions that further constrain the retrieval process to maintain specific latent code semantics. To this end, we make the following **contributions**:

- We analyze existing NLP methods for data augmentation and their suitability for code translation.
- Building on augmentation for bug-fixing, we propose rules for rule-based data augmentation in code translation.
- We explore the potential of the retrieval-based augmentation method with our proposed objective function that considers the translation retrieval accuracy and the consistency with the original dataset to effectively find new parallel code pairs.
- We compared various baselines to all the aforementioned methods. Our results suggest that both the rule-adaptations and retrieval-based methods are beneficial, but the latter outperforms the former significantly.

2 RELATED WORK

There is no existing work dedicated to training data augmentation for the supervised code translation task. Thus, we review related areas on program augmentation that is mainly used for the code repair application. Then we investigate data augmentation techniques that are prevalent in natural language translation.

2.1 Program data augmentation

The existing data augmentation methods for code mostly serve the application of code repair [5, 39]. The methodology is to create bug-fixing rules to generate more training data. Allamanis et al. created a more diversified and challenging dataset for the BugLab code repair model by augmenting their benchmark with mutated and mutilated programs generated by human-written rules [5]. These rules include changes to the original program's syntax, such as swapping the order of arguments in a function call or changing the name of a variable, or introducing small, intentional errors into

the code, such as typos or incorrect indentation. The more recent work on MultIPAs also uses rules to generate additional training data for the learning task of grading introductory programming assignments [39]. They expand on the rules introduced by Allamanis et al. [5] with six syntactic mutations that preserve the program's semantics and three semantic mutilations that introduce faults in the introductory programming assignments. As code translation aims at valid code fragments the methodology should avoid code mutiliation as done for augmenting code repairing datasets. To apply data augmentation to other tasks, such as method naming, code commenting, and code clone detection, Yu et al. leverage a pre-defined set of syntax-based rules to perform several program transformations to mutate programs written in Java [49]. All of these approaches are language and task-specific and require human expert efforts in generating rules. The performance heavily relies on the quality and quantity of these rules. Instead of manually creating rules, Liu et al. extract Java code snippets from Stack Overflow, and directly mine repair patterns from these code samples [29]. In our work, we introduce two novel code augmentation approaches tailored to the code translation task. One approach leverages specific rules for rule-based augmentation and inspired [29], the second approach leverages Big Code to mine translations.

2.2 Data augmentation for natural language translation

There are two types of methods for generating training data for natural language translation. The first is back-translation [13, 37, 45] and the second is word substitution [12, 20, 48]. Back-translation methods translate a sequence into the target language and then back into the original [45]. Specifically, they first use the current dataset to train a neural machine translation model. Then it obtains more mono-language data in the target language, which is usually a high-resource language, and uses the trained model to back-translate the data to the source language. Finally, these new pairs are added to the original dataset as augmentation. Chinea-Rios et al. select valuable target language instances from a large pool of source sentences, using embedding representations for back-translation [13]. Li et al. enhance back-translation with fuzzy string matches, improving neural machine translation's noise resistance while maintaining compact models [26]. Vaibhav et al. introduce a heuristic model for adding social media noise to text in labeled back-translation, showing that the right amount of noise enhances model robustness based on the dataset condition and model characteristics [47]. Li et al. use back-translation and self-learning to generate augmented training data [25]. Their method generates diverse synthetic parallel data on both source and target sides using a restricted sampling strategy during the decoding phase. Recently, Nguyen et al. proposed a method that is similar to back-translation [37]. Instead of using external mono-language data, they trained several forward and backward models and merged the original training data with the predictions of these models. While back-translation is the most common method for data augmentation in natural language, it is often vulnerable to errors in initial models. This is a common problem of self-training algorithms. Especially in code translation, this problem aggravates as parallel data is much more scarce.

Word substitution is an augmentation technique that replaces words in both source and target sentences with new vocabulary-matched words. Wang et al. introduce a randomized approach [48], while Fadaee et al. substitute low-frequency words in source sentences, predicting new words using contextually generated contexts [18]. Gao et al. employ a contextual mixture of related words to create the source sentence and randomly replace corresponding words [20]. Duan et al. propose a syntax-aware augmentation method, calculating word importance rather than random selection [17]. Cheng et al. use an adversarial network to generate new source sentences and implement interpolation with adversarial examples [12].
While the aforementioned methods work well for natural languages where there is more wiggle room with regard to noise, their application on programming languages is very risky as tiny mistakes can cause the failure of the program. In fact, code generated by these methods is often faulty or cannot be effectively used as training data.

3 FOUNDATION

In this section, we lay the necessary foundations for augmenting training data for supervised program translation. First, we give an overview of existing program translation methods, then we introduce code retrieval and motivate the methods applied in our work.

3.1 Program translation

The goal of program translation is to rewrite code in another desired programming language without changing the semantics. Consider the following small example of translating code from Java to C#: This is a Java code snippet:

```java
public class Example {
    public static void main(String[] args) {
        int x = 5;
        System.out.println(x);
    }
}
```

And this is the C# translation of the above snippet:

```csharp
using System;

public class Example {
    public static void Main() {
        int y = 5;
        Console.WriteLine(y);
    }
}
```

Both snippets implement the same function, but in C#, the name of `Main` method is capitalized, while Java uses lowercase `main`. In C#, `Console.WriteLine()` is used instead of `System.out.println()` in Java to output text to the console. And in order to use `Console.WriteLine()`, you need to include the `using System;`. The variable names are different in both programs but do not affect the semantics, and thus the translation is still valid.

3.2 Supervised program translation

There is plenty of work on supervised translation models, for example, Nguyen et al. utilized the phrase-based statistical machine translation (SMT) model to convert Java code to C# by applying it

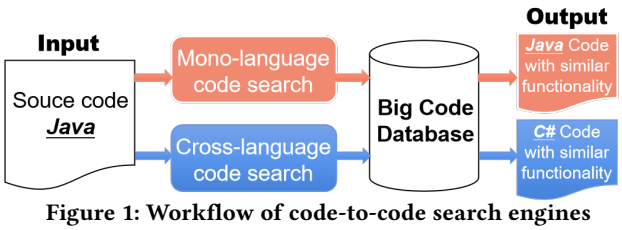


Figure 1: Workflow of code-to-code search engines

to the lexemes of the source code [34]. They later proposed a multi-phase, phrase-based SMT method that deduced and utilized both structure and API mapping rules [35]. However, these methods are restricted to languages that are either structurally or textually similar, such as Java and C#. To solve this problem, Chen et al. proposed a tree2tree model, which binary-encodes the code tree of a code fragment and translates it using an LSTM-based encoder-decoder model [11]. This model can be applied to more flexible types of languages and achieves higher accuracy on similar benchmarks [11]. As we only perform changes on the training dataset, not on the translation model, our methodology is model-independent. In our experiment, we choose the tree2tree model as our base model to test the augmentation as it is the state-of-the-art fully-supervised model.

3.3 Code retrieval systems

Since one of our augmentation methodologies is retrieval-based, we also give an overview of code retrieval techniques to set the context. In our translation uses-case, we have a code corpora at our disposal, we need a code-to-code search system in contrast to description-to-code appraoches that require user annotations [27]. There are mono-language and cross-language code-to-code retrieval systems. An example of a mono-language search system is Facoy [23]. Mono-language systems aim at finding code that implements the same functionality in the same language repository. For example, if the input code is written in Java, the search engine will retrieve another Java code with similar semantics (red path in Figure 1 is the simplified workflow). In cross-language search, the goal is to find code snippets that implement the same functionality in a different language. For example, if the input code is written in Java, the retrieved code should be written in another target language such as C# (blue path in Figure 1). Examples of such systems are YOGO [41],COSAL [33], RPT [9] and BigPT [8]. RPT can also be used as a mono-language system.

In our augmentation system, we leverage both, a mono-language search engine to find similar code to the source code to limit the dataset scope, and for the corresponding translation, we require a cross-language search engine to retrieve the translation in the target language. Since we want to perform the augmentation with the least human effort and make the most of the existing resources, we need code-to-code search engines that are fully automatic and only need Big Code resources as input. This eliminates FaCoY and YOGO from our choices. FaCoY necessitates Q&A posts from Stack Overflow as query supplements [23]. YOGO provides a cross-language graph representation, but it necessitates handwritten semantic rules and patterns in a domain-specific language for each query [41]. Recent work COSAL proposes a cross-language technique that uses both static and dynamic analyses to identify similar code and does not require a machine learning model [33]. However, it cannot be easily generalized to other languages except Java and Python, which have been studied in their work. The fully-automatic search engines RPT [9] and BigPT [8] both can serve our retrieval-based augmentation methods, as they only require raw code as the search query. They are both language-independent. Therefore, in our work, we use RPT for mono-language retrieval and BigPT for cross-language retrieval out of convenience. Both are treated as black boxes in our approach.

4 METHODOLOGY

In this work, we address training data augmentation for supervised code translation. The input to our approach is a parallel training dataset for a code translation model. Each data point in the dataset is a pair of code snippets such that one snippet is in the source language and the other is its translation in the target language. Our goal is to find new data points based on the training dataset to augment it so that the performance of the code translation model can be improved. To achieve this, we have to augment the parallel dataset in a way that unseen relationships between source and target language are introduced to the model, while the general distribution of data points is not significantly shifted.

To address this problem we explored two different avenues inspired by natural language translation: first, we investigate rule-based methods (Section 4.1) and adapt standard word masking and back-translation to code-specific tasks. We suggest a set of code transformation rules specific to code translation by modifying other existing ones that are used for code fixing [5, 39]. These rules make slight changes to the tokens and structures of a program. Given the fact that the rule-based method requires human interaction to define rules and limits the type of augmentation, we further explore data augmentation through code retrieval (Section 4.2). Due to the large quantity and scope of the online big code resources, we propose to distill them for additional snippet pairs. This distillation should lead to data points that enrich the original training set with novel unseen relationships while preserving the distribution of the original training set.

4.1 Rule-based data augmentation

The classic and straightforward way of augmentating data is to create rules. Code transformation rules can be applied to any existing data point to deterministically obtain new points from the current dataset. As no prior set of transformation rules has been created specifically for the program translation task, we created them with the following set of requirements:

(1) the rules have to be simple and automatically applicable to the code avoiding manual inspection.
(2) the rules should be designed specifically for code translation. Unlike existing classification tasks for code repair, code translation is a task that uses a parallel dataset where input and output are both program data. And the goal of a translation model is to learn the mapping between two languages. There are already some attempts on data augmentation for bug detection [5, 39]. Some of their rules are to randomly add bugs to the code, which cannot be directly used in the code translation use case because it creates non-functional code, which is also not easy to map

in a different language. Thus, the rules have to be designed in a way so that they are transferrable from the source to target language or vice versa.

(3) the rule-generated code should be semantically functional. If the code snippets do not semantically make sense, they cannot be regarded as programming language text, which will not be suitable for training a program translation model.

(4) the rules should bring variety to the original dataset so that the model can learn new relationships between the two languages. For example, some rules in the bug detection augmentation are swapping the if-else branch and declaring dummy variables. Both are unnecessary in the context of translation as the original dataset typically already captures such relationships.

(5) the rules should not be language specific so that they can be applied in arbitrary language pair settings (imperative languages). Thus, we avoid rules that involve specific grammar only in certain languages.

Based on all the aforementioned requirements, we took an existing rule set designed for augmenting training data for bug detection [5, 39] and adapted them to the use case of code translation. First, we removed all rules that aim at introducing bugs, as in code translation we want to ensure the model learns valid translations of functional code. We further overtook and merged similar types of rules, such as operator change and operator mirroring. Finally, we added novel rules that deal with code block splitting and program level merging. All in all, we create the following four data augmentation rules for the code translation task:

(1) Reverse: We reverse the logic of a conditional statement, e.g. $>$ to $\leqslant$, == to ! =, `true` to `false`, etc. Then we change the corresponding statement in the code snippet in the target languages. We only make revisions on the conditional statements because the parser tool can identify the corresponding statement easier and more accurately than going through the whole snippet. We use the mutation operators to implement this rule in the experiment [22].

Example1:

```
// original code
if (x == 5)

// transformed code
if (x != 5)
```

(2) Merge: We implement the merging rules on two levels:

- Control structure level: we merge the content for conditional statements. When there are two conditional statements in the code snippet, we merge them into one statement using the && operator.

Example2:

```
// original code
if (x > 5) { i ++; }
if (y < 10) { System.out.println("pass"); }

// transformed code
if (x > 5 && y < 10) {
    i ++;
    System.out.println("pass");
}
```

- Program level: we merge two code snippets into one by simply concatenating them. We only implement this rule on the code snippets that have fewer than 10 lines so that the complexity of the code snippets is not increased significantly. Then we concatenate the corresponding parts in the target code snippet.

(3) Split: the reverse of structure level *Merge* rule, we split a conditional statement if there are two conditions connected by &&, then change the corresponding target code snippet.

(4) All: the combination of the above three rules. We collect the code generated by any of the aforementioned rules.

These rules are specifically designed for programming languages, but general to common imperative languages. Thus, they can be applied to both the input code and its translation, maintaining the logical semantic between source and target.

4.2 Retrieval-based data augmentation

In addition to the rule-based method, we explore the retrieval-based data augmentation method that leverages Big Code and develop and advanced approach. Our assumption is that the large code repositories contain sufficiently interesting code fragments that can complement existing training datasets. For retrieval-based augmentation, we directly search for code pairs from the big code databases using mono- and cross-language code search tools. The retrieved code pairs will be added to the original dataset as augmented data. To protect the distribution of the original dataset, we do not randomly pick data points from the big code database. Instead, we aim to find translation pairs that not only introduce new information to the model but are also similar to data points of the original training dataset. Thus, for each data point in the original training data, we first search with a mono-language code-to-code retrieval engine for a code snippet similar to the code fragment in the source language. Then, we use a cross-language retrieval engine to find a potential translation in the target language. The resulting pair will be used to augment the original dataset. This way, no explicit rules need to be created.

To implement such a data augmentation strategy, we first need code retrieval tools that can perform both mono-language retrieval and cross-language retrieval to search for potential translation. Further, we need optimization criteria that select the most suitable candidates among retrieved solutions. In this section, we first show the workflow of a naive retrieval-based method without optimization criteria for selecting candidates. After that, we discuss our optimization criteria and our improved retrieval-based methodology in detail.

4.2.1 Workflow of the naive retrieval-based methodology. Our goal is to find new snippets in source and target language from Big Code. The input is the training dataset of a code translation model. The dataset consists of a set of translation pairs: (A_1, B_1), (A_2, B_2), ... For any pair (A, B), A is a code snippet written in the source language, and B is its translation in the target language.

In our implementation, we directly leverage the existing mono-language code search tool [9] and cross-language code search tool [8] to formulate a retrieval-based augmentation pipeline for the code translation model. In the naive workflow, we greedily simply fetch the top one output of these tools without any modification.

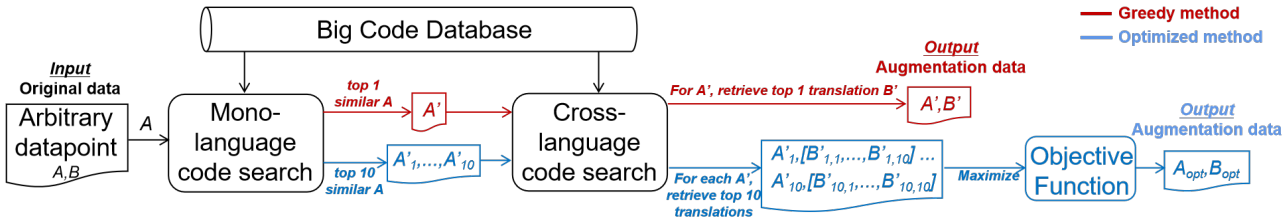
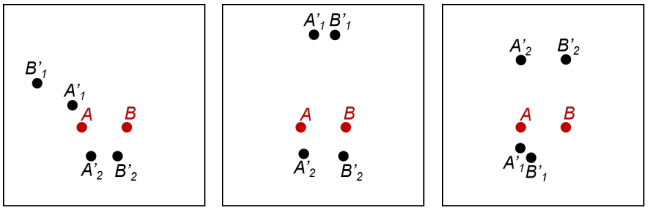


Figure 2: Workflow of retrieval-based augmentation for code translation



(a) Greedy successive selection of source and target.

(b) Pick the pair with the most promising translation

(c) Pick pairs where source and target are equidistant to the original point

Figure 3: Identify the optimal augmentation data

The workflow is shown as a red pipeline in Figure 2. The approach consists of two steps. First, to preserve the original data distribution, we use a mono-language code search tool to retrieve a code snippet similar to source code A from the big code database. By default, this process returns the top-1 similar code snippet A'. Second, we find a code snippet that can be regarded as the translation of A' (but written in the target language) using a cross-language search tool. We denote it as B' and consider it a translation of A'. The result of this process is a pair of code samples (A', B') that is similar to the original pair (A, B). We add the new pair to the original dataset to generate the augmented dataset.

To augment the dataset with multiple code pairs (A_1, B_1), (A_2, B_2), ..., we apply the aforementioned protocol to a random sample. The amount of data points we pick depends on the augmentation ratio we want to obtain, i.e. the ratio of the original data and the augmentation data. For example, in the default setting of most scenarios, the ratio is $1:1$, i.e., one augmented point per existing training data point.

Limitation of the naive workflow. In the naive workflow, we only select the local optimum based on the existing retrieval systems. In Figure 3a, (A'_1, B'_1) would be the solution of the naive approach, i.e., A'_1 is most similar to A and B'_1 is the best translation for A'_1. However, (A'_1, B'_1) might not be the most suitable pair. B'_1 might be just by chance the best possible translation for A'_1, but for a different A'_i, we might be able to find a more accurate translation B'_i Further, the similarity of B'_1 might diverge more from B than A'_1 does from A. Such a point might shift the distribution of the training dataset and introduce noise. In contrast considering (A'_2, B'_2), while A'_2 is less similar to the original A compared to A'_1, the pair might be more suitable because its similarity to A resembles the similarity between B and B'_2 and B'_2 is a more accurate translation for A'_2 than B'_1 is for A'_1.

Following the previous thoughts, there are two more heuristics to consider to find the optimal pair. For example, in Figure 3b, (A'_1, B'_1) is the most accurate translation as the distance between A'_1 and B'_1 is smallest among all pairs. But it is too far away from the original translation pair (A, B). The translation pair (A'_2, B'_2) could be an alternative choice. Although the translation is not as accurate as (A'_1, B'_1), it better matches the characteristics of the original dataset. Thus, we need to also capture this aspect when selecting the optimal pair.

A simple way to account for this is to consider the similarity between A and A', and the similarity between B and B'. However, in this approach, we would only consider the local similarity for an individual A and individual B, ignoring their fit as a pair. Theoretically, A' and B' could entirely diverge from each other. Thus, we also need to make sure the code pair as a whole is similar to the original pair. For example, in Figure 3c, A'_1 is most similar to A, B'_1 is the most similar to B. However, the similarity of A and A'_1 is very distinct from the similarity between B and B'_1. While A'_2 and B'_2 are less similar to A and B respectively, the relative distance between A and A'_2 resembles the distance between B and B'_2. Thus, it is more likely that (A'_2, B'_2) can better align with the original data, i.e., we can regard the (A'_2, B'_2) as a pair that is more similar to the original pair (A, B). One can estimate this divergence by the difference between the two similarity vectors.

To address the aforementioned cases, we introduce a new workflow that considers this spectrum of candidates and uses an optimization function to select the most promising data point for augmentation.

4.2.2 Workflow of the optimized retrieval-based methodology. We consider a starting dataset $D_0 = (A_i, B_i)$ code samples A_i in the source language and B_i in the target language. We aim to augment this original dataset with appropriate tuples extracted from an external dataset D_{aug}, which contain tuples in source and target language, correspondingly.

Our approach has a two-stage workflow for generating an individual augmented tuple: (1) For a tuple $(A, B) \in D_0$, we generate the set of potential candidates for augmentation obtained from D_{aug}(2) we solve an optimization objective to find one pair of optimal code snippets for augmentation.

The blue pipeline in Figure 2 shows the workflow for generating the initial set of candidates. In contrast to the greedy approach, which only keeps the top-ranked output of a search as an augmentation candidate, we use the mono-language code search tool to retrieve top-K results $A'_1, ..., A'_K$ to build the candidate set. Then for each A'_i, we use the cross-language code search tool to retrieve the top-K translations for it. In the end, we obtain $K \times K$ translation pairs $(A'_1, [B'_{1,1}, ..., B'_{1,K}]), ..., (A'_K, [B'_{K,1}, ..., B'_{K,K}])$ which we denote as $D_{candidate}$. In our experiments, we set K to 10, so we obtain 100 translation pairs as candidates for each sample (A, B) taken from

the original dataset. Users can set the K to their preferred value based on their computation resources and their needs.

4.2.3 Objective functions for selecting the optimal translation pair. Now, we discuss how we select the optimal translation pair from the $K \times K$ candidate space D_{match}. Starting off with a code pair $(A, B) \in D_0$ that defines the context for a new augmentation tuple, our goal is to select a tuple (A', B') from the candidate set $D_{candidate}$ (generated as discussed in Section 4.2.2) such that the two pieces of code A' and B' are very good translations of each other, and each individual piece of code shares the same commonalities with its counter part from the original sample A and B. In other words, if A' is different in some aspects from A, we want B' to differ in the same manner and not completely diverge.

For this purpose, we need to assess the distance or similarity between each individual code snippet A, B, A', and B' with a universally usable similarity function $S : D \times D \rightarrow \mathbb{R}$ where $D = D_0 \cup D_{aug}$ for code pieces across languages. Here, we made a practical choice by picking the similarity function implemented in BigPT [8], which considers structural and textual aspects of code snippets for its calculation. We use this similarity function, denoted as S in three different ways to define three independent optimization criteria that fulfill our requirements.

Considering the context $(A, B) \in D_0$ and a sample from a search space $(A', B') \in D_{candidate}$ we consider

(1) $f_1(A', B') = S(A', B')$ as a measure of similarity within the new pair,
(2) $f_2(A', B') = S(A', A) + S(B', B)$ as a measure of similarity between the new tuple and context and
(3) $f_3(A', B') = -|S(A', A) - S(B', B)|$ as an alternative similarity measure within the new pair. Intuitively, we aim to find examples where A is just as similar to A' as B is to B'.

Given the three objective functions that map the search space into $\mathbb{R}$, we can formalize our task as a multi-objective optimization problem given by the following statement

$$\mathbf{max} f_1(A', B'), \mathbf{max} f_2(A', B'), \mathbf{max} f_3(A', B'),$$
$$A' \in D_{aug}, B' \in D_{aug}$$

We rely on optimizing a scalarized objective $f_{obj} = \alpha f_1 + \beta f_2 + \gamma f_3$ where α, β and γ are set to 1 by default. We have tested different methods for scalarization $\alpha, \beta, \gamma \in \{0, 1\}$ to evaluate the benefits from each component with the experiments discussed in Section 5.2.

Example 3: A translation pair from Java to C# in the original training set:

```
//---Java code---
private void createTypedPrimitiveArray ( ItemArrays item
    ) {
    item._typedPrimitiveArray = new int [ data.length ] ;
        }
//---C# translation---
private void CreateTypedPrimitiveArray (
    IntHandlerUpdateTestCase.ItemArrays item ) {
    item._typedPrimitiveArray = new int [ data.Length ] ;
        }
```

Based on this pair, our method retrieves a similar pair for augmentation:

```
//---Java code---
private void createTypedPrimitiveArray ( ItemArrays item
    ) {
    item._typedPrimitiveArray = new double [ data.length
        ] ; }
//---C# translation---
private void CreateTypedPrimitiveArray(
    FloatHandlerUpdateTestCase.ItemArrays item) {
    item._typedPrimitiveArray = new float[item.data.
        Length]; }
```

This new pair is similar to the original data, yet it cannot be generated by any of rules presented in Section 4.1 as it would require knowledge of how different data types can be mapped across programming languages without introducing errors. The model can now learn the relationship between the types double and float, and that float is more similar to double than integer. Mixing up floats and integers is a common translation mistake in existing translation models [35] and can be avoided with such additional data points.

Table 2: Statistics of Java to C# dataset [11]

Dataset	Lucene	POI	IText	JGit	JTS	ANTLR
# of data pairs	5516	3153	3079	2780	2003	465

5 EXPERIMENTS

We conducted several experiments to evaluate the effectiveness of all mentioned data-augmentation strategies for the supervised code translation task. As there are no existing data augmentation methods for supervised code translation, we compare the rule-based approaches and the optimized retrieval-based methodology **CTAug** (Data **Aug**mentation for **C**ode **T**ranslation Model) to other NLP baselines. To explore more possible strategies to implement the retrieval-based methodology, we also conducted several micro-benchmarks by changing the objective functions of the retrieval-based method, the augmentation modes, and the amount of augmented data. In the end, we discuss the limitations. In our experiments, we use the state-of-the-art model Tree2tree [11] as the underlying code translation model.

Datasets. For our experiments, we use two different types of datasets. For training and testing the model, we directly use the datasets from the tree-to-tree code translation work [11]. These datasets are also used in other program translation papers [34, 35]. They were originally built based on several authoritative open-source projects, which have both official Java and C# versions. Similar to the related work we consider the task of translating Java to C# in order to keep consistency to baseline. However, our methodology is language-independent, and the code retrieval systems we implement in our experiment (RPT [9] and BigPT [8]) can also apply to multiple languages. The statistics of these datasets are shown in Table 2.

The second dataset is the big code database that we use to retrieve similar code in both the same and different languages. Similar to prior work [8], we generate the database from the Public Git Archive (**PGA**) - a dataset with more than 260,000 top bookmarked Git repositories from GitHub [3, 32]. We fetch the projects written in Java and C#, and clean them by removing duplicate and corrupted files. In addition, we also remove the projects that occur in the **Java-C#** dataset to avoid the information leak. Finally, we obtain a 68GB dataset with 5,891 projects and around 2,691 functions for each program. We generated the program representation for each code snippet and indexed the data using BigPT [8].

Default setting of CTAug. In the default augmentation mode, we implement the system in a straightforward way: CTAug first retrieves similar source code from the big code database, then retrieves its translation (we also evaluated other modes in Section 5.3). We use the multi-objective optimization function f_{obj} mentioned in Section 4.2.3 as the default objective because it addresses all possible optimizations (we also evaluated other objective functions in Section 5.2). By default the augmentation ratio is 1:1 as suggested in prior work [20, 25, 48] (Section 5.4).

Metrics. We use the following metrics that have been traditionally used for measuring the quality of program translation [11, 35]:

- **Program accuracy** (PA) is the percentage of the predicted translation that is the same as the ground truth. PA is an underestimation of the true accuracy based on semantic equivalence. It does not account for programs that only differ in writing habits and style. This metric is more meaningful than other previously proposed metrics, such as syntax-correctness and dependency-graph-accuracy, which are not directly comparable to semantic equivalence [8, 11, 35]. We could not use computational accuracy [43] because the ground truth data is not always executable without its entire project context.
- **Token accuracy** refers to the percentage of the tokens that are exactly the same as the ground truth [11]. We measure it as in the Tree2tree work [11] to provide some additional insights into the performance.
- **CodeBLEU** is the BLEU score adaption for code, which is widely used in code synthesis [42]. It includes the original n-gram match in BLEU, and further injects code syntax via abstract syntax trees and code semantics via data-flow.

5.1 Comparison with baselines

We evaluate the effectiveness of CTAug on the datasets from Table 2. For each code pair in the dataset, we retrieved an augmented pair from the big code database and add the new pair to the original dataset. Then we train the tree-to-tree code translation model on the augmented dataset. For each experiment, we apply ten-fold validation on matched pairs. We compare CTAug to adapted natural language augmentation methods and our rule-based approach.

5.1.1 Natural language augmentation methods. We implement the following two mainstream methods as our **baselines**:

(1) Word masking (WM): word masking is one of the most used basic word-level augmentation methods. It randomly removes a word in a sentence to create a new sentence as the augmentation data. We implement this method on both source and target code with the assistance of the text data augmentation tool nlpaug [31] to generate new data as augmentation data. Considering the length of each line in the code snippet, we regard two lines as a natural language sentence. Note that, it is different from the word masking used in unsupervised translation, which aims to recover the masked word in mono-language to improve the language model for one specific language.

(2) Back-translation (BT): back-translation is the most widely-used data augmentation method on natural language translation tasks. Therefore, we attempt to implement it on code as a baseline comparison. We train the tree-to-tree model on the original Java to C# dataset in the reverse direction, i.e. C# code is the input, and Java code is the output. Then we randomly pick the C# code from the big code database **PGA**, then use this trained model to back translate these C# code snippets to Java code. We use these newly matched Java-C# code pairs as the augmentation data to be added to the original dataset.

Table 3: Performance comparison with the baselines

Dataset	w/o Aug	CTAug	NLP Methods		Rule-based methods			
			WM	*BT*	*Reverse*	*Merge*	*Split*	*All*
Metric: Program Accuracy								
Lucene	72.8%	**85.8%**	63.1%	70.2%	72.7%	72.0%	72.8%	75.9%
POI	72.2%	**86.1%**	62.4%	69.9%	73.6%	70.3%	72.5%	75.1%
IText	67.5%	**83.2%**	61.5%	66.3%	68.9%	66.9%	67.8%	72.8%
JGit	68.7%	**81.7%**	59.3%	65.4%	70.3%	70.2%	69.1%	72.8%
JTS	68.2%	**82.3%**	61.2%	67.7%	70.9%	69.8%	69.6%	73.3%
ANTLR	31.9%	**66.2%**	25.3%	31.9%	33.6%	36.7%	32.7%	40.1%
Metric: Token Accuracy								
Lucene	85.3%	**92.3%**	80.3%	87.7%	88.3%	87.1%	88.9%	90.1%
POI	84.8%	**94.8%**	81.1%	78.2%	87.2%	85.2%	87.3%	91.7%
IText	80.3%	**93.3%**	77.2%	83.1%	81.3%	79.8%	79.2%	85.6%
JGit	81.7%	**88.9%**	73.7%	82.3%	84.2%	82.5%	82.5%	88.2%
JTS	82.1%	**90.2%**	72.1%	82.8%	83.5%	81.2%	83.6%	89.7%
ANTLR	70.2%	**80.1%**	57.2%	69.2%	73.8%	75.7%	74.9%	78.3%
Metric: CodeBLEU								
Lucene	87.6%	**95.3%**	84.4%	91.2%	92.8%	89.5%	92.6%	94.2%
POI	88.6%	**96.7%**	85.4%	81.7%	90.0%	88.9%	91.3%	94.5%
IText	84.8%	**96.5%**	81.3%	87.8%	85.6%	83.5%	84.3%	89.2%
JGit	85.1%	**92.4%**	78.9%	86.3%	88.7%	87.2%	85.9%	91.6%
JTS	86.7%	**93.6%**	75.7%	87.1%	88.0%	84.1%	85.8%	93.1%
ANTLR	75.2%	**83.9%**	62.2%	73.3%	78.1%	80.1%	77.6%	82.8%

5.1.2 Rule-based baselines. As discussed in Section 4.1, we defined four revision rules for code. For each code snippet, we tested the application of each individual rule and generated a data point upon success. We use ANTLR [1] to parse the generated code so that we can remove those with incorrect grammar. Then we augment the training dataset with the remaining ones. In our experiments, the amount of generated augmented data by only *Reverse* rule is 46%. We obtain 73% using both *Merge* rules and 22% using the *Split* rules. As a result, in the *All* augmented dataset, the ratio of original data and augmented data is around 1:1.4.

Results. The performance of the baselines is shown in Table 3. The first table shows the PA of each baseline. CTAug significantly enhances the original model by increasing the accuracy by ~15%. For the ANTLR dataset, the improvement even reaches ~35% because the original dataset is very small and the data augmentation contributes a large number of training data.

Both natural language baselines do not perform well for programming language augmentation. The application even harms the results in some cases. Augmentation with word masking results in 5-10% performance loss. As previously discussed, simply masking one word can harm the whole structure of the code and damage the code. As a result, the model produces more wrong outputs. The back-translation model performs better than word masking as it leverages real human-written C# code from the database. However, it still harms the results by up to 5%. We observed that the code snippets that cannot be correctly translated by the original model mostly cannot be correctly translated after augmentation too. Therefore, the performance of the back-translation method could be affected by the vulnerability to the quality of the original model. Having a high-quality initial translation model is difficult because of the rather small size of the initial training dataset [11]. In conclusion, straightforward adaptation of natural language data augmentation methods on code data is not effective.

Table 4: Ablation study on objective functions

Dataset	w/o Aug	Objective			
		Global_1 (default)	*Global_2*	*Global_3*	*Greedy*
Lucene	72.8%	**85.8%**	77.1%	82.2%	75.6%
POI	72.2%	**86.1%**	79.8%	83.6%	78.3%
IText	67.5%	**83.2%**	74.3%	78.8%	71.2%
JGit	68.7%	81.7%	**82.1%**	76.9%	69.9%
JTS	68.2%	**82.3%**	75.0%	79.5%	72.8%
ANTLR	31.9%	**66.2%**	52.5%	60.3%	42.2%

Augmentation with rules only slightly improves the performance of the model. For *Reverse*, *Merge*, and *Split* rules, the performance is similar to the original dataset. One possible reason is they do not contribute additional new data. The *Split* rule generally only contributes 22% more training points, which explains why it performs the worst. Another reason might be that these rules do not introduce new information into the model. The rules might already have been learned by the model with the existing training data. In some cases, these rule-based methods might even harm the results, for example, *Merge* rule on the POI dataset. The possible reason is that repetition of similar training instances causes overfitting during training so the model cannot generalize well. Putting all rule-based results together (*All*) there is a visible performance gain which still is far below the retrieval-based method.

In the following subtables of Table 3, we also show the token accuracy and the CodeBLEU score of each method. We can see that although the scores are different from program accuracy, the retrieval-based augmentation method still outperforms all other methods. The token accuracy shows that the results are satisfactory with regard to the textual similarity. The good performance on CodeBLEU further shows the augmented model can also address the syntax and semantics of the code.

5.2 Comparing different objective functions

As discussed in Section 4.2, we implement an objective $f_{obj} = \alpha f_1 + \beta f_2 + \gamma f_3$ to select the globally optimal data pairs for augmentation. We considered three objective functions together to ensure translation quality, preservation of the original data distribution, and data variety. To show the impact of each individual objective, we conduct an ablation study by changing the scalarization parameters α, β, and γ. In the default CTAug, we set all parameters to 1. Therefore, the default objective is:

- Global_1 (default): $f_{obj} = f_1 + f_2 + f_3$

For the ablation study, we keep the first objective $f_1(A', B') = sim(A', B')$ since the accuracy of the translation is always essential. As a result, we experiment with the following objectives:

- Global_2: $f_{obj} = f_1 + f_2 = sim(A', B') + sim(A', A) + sim(B', B)$ ($\alpha = 1, \beta = 1, \gamma = 0$). In this objective, we only consider the quality of the translation of the new pair, and the similarity between each new code snippet and the original one. We abandon the third objective.
- Global_3: $f_{obj} = f_1 + f_3 = sim(A', B') - |sim(A', A) - sim(B', B)|$ ($\alpha = 1, \beta = 0, \gamma = 1$). In this objective, we leave out the second objective

In addition, we also test the greedy strategy that does not use any of these objectives.

Result. Table 4 shows that in all except one case, the default objective function outperforms all the other variations. The accuracy difference between the default objective and the Global_2 objective is within 10%. The accuracy difference between the default objective and the Global_3 objective is within 5%. Thus $f_3(A', B') = -|sim(A', A) - sim(B', B)|$ has a higher impact than $f_2(A', B') = sim(A', A) + sim(B', B)$. The possible reason is that we already performed pre-selection using the retrieval tools to select the top 100 translation pairs. The retrieval tools guarantee the similarity between the A and A' and the similarity between the A' and B'. Therefore, $sim(A', A)$ is already maximized during the pre-selection, removing this component from the objective does not affect the result dramatically. Since our results are not deterministic but probabilistic, there could be some minor deviations. For example, on the JGit dataset, the result of Global_2 is 0.4% higher than the default objective.

Furthermore, we can also see that greedy performs in general worse than any variation of global optimization. This result shows that it is worthwhile to consider the similarity alignment of the translation pairs from all three similarity perspectives. The optimal choice of α, β, and γ varies from case to case that is based on the supervised model or the dataset properties. In practice, user can fine-tune these hype-parameters using optimization strategies such as grid search.

Table 5: Performance of different augmentation modes

Dataset	New similar source		Existing source	
	A **first**	B **first**	A' **first**	B' **first**
Lucene	85.8%	**87.1%**	74.5%	75.6%
POI	86.1%	**86.3%**	68.1%	73.1%
IText	83.2%	**84.5%**	72.2%	68.9%
JGit	**81.7%**	80.2%	67.3%	66.7%
JTS	82.3%	**83.7%**	69.5%	68.2%
ANTLR	66.2%	**69.9%**	35.7%	37.1%

5.3 Different starting points for augmentation

We further conducted experiments to verify whether our approach for the initial choice of A' matters. Therefore, we implemented different starting modes as explained in the following. Let D_o be the original training dataset and D_{big} the big code database. Let $(A, B) \in D_o$ be an arbitrary pair. The four starting modes used in our experiment are:

- **Similar A'**. In our default system, we pick the code snipped in the source language A from the original dataset D_o, then find a new code snippet A' similar to A from D_{big}. Then we use A' as a query to retrieve its translation B', which is written in the target language, also from D_{big}.
- **Similar B'**. We also implement the aforementioned approach starting with B first. Then, we find a new code snippet B' similar to B from D_{big}. Then we use B' as a query to retrieve its translation A' from D_{big}.
- **$A = A'$**. Here, we take A and directly retrieve a translation B' from D_{big}, obtaining the pair (A, B').
- **$B = B'$**. Similar to the third mode, we take B and directly retrieve a translation A' from D_{big}, obtaining the pair (A', B).

All other parameters remain as defined for CTAug.

Results. The performance of different modes is shown in Table 5. First, we can see the model trained with entirely new pairs (A', B') achieves the best result. And both directions A' and B' first improve the performance, similarly. In most cases, retrieving similar target code first leads to slightly better performance. The possible reason is that the cross-language search method performs better on the C# to Java case than the Java to C# case, as shown in the BigPT paper [8]. When existing snippets A/B are directly used for

Table 6: Impact of the amount of augmentation data

Dataset	Original:Augmentation			
	1:0.5	1:1	1:1.5	1:2
Lucene	83.2%	**85.8%**	84.6%	80.6%
POI	83.1%	**86.1%**	84.8%	77.1%
IText	79.6%	**83.2%**	81.9%	72.3%
JGit	77.7%	**81.7%**	80.2%	74.9%
JTS	78.5%	**82.3%**	80.3%	74.5%
ANTLR	63.8%	66.2%	**66.8%**	60.2%

alternative translations B'/A'„ we observe rather erratic results. In some cases, they improve the results of the original model by up to 6%. In other cases, they harm the performance by up to 5%. It is possible that the one-to-many matches confuse the model.

5.4 Impact of the augmentation ratio

The ratio of the original data and the augmentation data can also affect the augmentation performance. In our default setting, we use the 1:1 ratio that is commonly used in existing work [19, 48]. To find out the impact of the ratio, we also test the ratios, 1:0.5, 1:1.5, and 1:2. For the ratio 1:0.5, we randomly pick half of the data from the original dataset to retrieve the data points from the big code as the augmentation data. For the ratio 1:2, we keep the second best-retrieved result to double the number of augmentation data. For the ratio 1:1.5, we randomly pick half of the data and also keep the second-best result for each.

Results. Table 6 shows the results for each dataset and augmentation ratio. Except for the Antlr dataset, the 1:1 ratio leads to the best performance. From a ratio 1:0.5 to 1:1, more augmentation data introduces more information to the model and improves the performance. However, further increasing the augmentation data diminishes the performance. The first reason might be there are not enough diverse data points introduced, i.e. there are too many similar data points. Having an abundance of similar training examples can cause the model to become overly specialized in capturing the specific variations and noise present in the training set. As a result, the model might learn to memorize the training data rather than learn the underlying patterns. The second reason might be the fact that for each original data point, we resort to the second-best point based on our scores and introduce more noise.

This experiment shows that there is an optimal ratio of augmentation data when using our method. The optimal ratio can differ between different datasets. For the datasets used in our experiment, 1:1 in general is a good choice. With sufficient computation resources, one can also use grid search to find the optimal ratio.

5.5 Limitations

Since our goal is making the most of the existing resources without making extra efforts, the augmentation performance relies on the quality of the original dataset, the big code database, and the code retrieval techniques. For example, here is one failed case of CTAug:

```
//---Input: Java code---
public class IndexWriter {
  private final IndexWriterConfig config;

  public IndexWriter(Directory directory,
      IndexWriterConfig config) {
    this.config = config;
    //...}
//...}
```

```
//---Ground truth: C# translation---
public class IndexWriter : IDisposable{
    private readonly IndexWriterConfig _config;

    public IndexWriter(Directory directory,
        IndexWriterConfig config)
    {
        _config = config ?? throw new
            ArgumentNullException(nameof(config));
        //...}
//...}
```

```
//---Output: Incorrect C# translation---
public class IndexWriter {
    private IndexWriterConfig _config;

    public IndexWriter(Directory directory,
        IndexWriterConfig config) {
        _config = config;
        //...}
//...}
```

The model failed to translate the use of `readonly` and the presence of the `IDisposable` interface in the C# version because they do not appear in the Java version and the model has limitations in learning language-specific features and conventions. The model also failed to check if the `config` argument passed to the constructor is `null` because, in the Java version, the `config` argument is already a required parameter and is not nullable. The model cannot address the different contexts, and it cannot find augmentation data from the Big Code database to complement this missing information based on the existing training data.

Furthermore, CTAug also has limitation usage-wise. It is a tricky and exhausting process to find the optimal hyper-parameters. We need to retrain the model for each hyper-parameter combination, which takes a lot of time.

6 CONCLUSION

In this work, we explored the data augmentation methods for supervised code translation. We evaluated rule-based and retrieval-based data augmentation strategies that are specifically designed for code translation task. The latter, which is most effective, retrieves similar translation pairs based on the original dataset by leveraging mono-language and cross-language code retrieval techniques. To select suitable data pairs for augmentation, we designed objective functions to preserve the data distribution and increase the data variety. Our ablation study shows the effectiveness of the selection mechanism. The task-specific rule-based method performs worse than the retrieval-based method, but outperforms vanilla adaptations of natural language augmentation methods. The experiments show that the retrieval-based method can significantly improve data performance without much human intervention.

Future work. First, one can try to combine different augmentation strategies, such as rule-based and retrieval-based techniques, to maximize the benefits. Further, we believe that code augmentation through constrained retrieval can also be helpful in other supervised code-learning tasks, such as code stylization [10] or code summarization [4].

REFERENCES

[1] 2019. ANTLR. https://www.antlr.org/. [Online; accessed 28-Apr-2023].

[2] 2019. Java2CSharp. https://sourceforge.net/projects/j2cstranslator/. [Online; accessed 28-Apr-2023].

[3] 2019. Public Git Archive. https://github.com/src-d/datasets/tree/master/PublicGitArchive. [Online; accessed 28-Apr-2023].

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 4998–5007.

[5] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 27865–27876.

[6] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*. JMLR.org, 173–182.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[8] Binger Chen and Ziawasch Abedjan. 2021. Interactive Cross-language Code Retrieval with Auto-Encoders. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 167–178.

[9] Binger Chen and Ziawasch Abedjan. 2021. RPT: Effective and Efficient Retrieval of Program Translations from Big Code. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 252–253.

[10] Binger Chen and Ziawasch Abedjan. 2023. DuetCS: Code Style Transfer through Generation and Retrieval. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE / ACM.

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*. 2547–2557.

[12] Yong Cheng, Lu Jiang, Wolfgang Macherey, and Jacob Eisenstein. 2020. AdvAug: Robust Adversarial Augmentation for Neural Machine Translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 5961–5970.

[13] Mara Chinea-Rios, Álvaro Peris, and Francisco Casacuberta. 2017. Adapting Neural Machine Translation with Parallel Synthetic Data. In *Proceedings of the Second Conference on Machine Translation, WMT 2017, Copenhagen, Denmark, September 7-8, 2017*. Association for Computational Linguistics, 138–147.

[14] Alexis Conneau, Ruty Rinott, Guillaume Lample, Adina Williams, Samuel R. Bowman, Holger Schwenk, and Veselin Stoyanov. 2018. XNLI: Evaluating Cross-lingual Sentence Representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 2475–2485. https://doi.org/10.18653/v1/d18-1269

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. Association for Computational Linguistics.

[16] Terrance Devries and Graham W. Taylor. 2017. Improved Regularization of Convolutional Neural Networks with Cutout. *CoRR* abs/1708.04552 (2017).

[17] Sufeng Duan, Hai Zhao, Dongdong Zhang, and Rui Wang. 2020. Syntax-aware Data Augmentation for Neural Machine Translation. *CoRR* abs/2004.14200 (2020).

[18] Marzieh Fadaee, Arianna Bisazza, and Christof Monz. 2017. Data Augmentation for Low-Resource Neural Machine Translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*. Association for Computational Linguistics, 567–573.

[19] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard H. Hovy. 2021. A Survey of Data Augmentation Approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*. Association for Computational Linguistics, 968–988.

[20] Fei Gao, Jinhua Zhu, Lijun Wu, Yingce Xia, Tao Qin, Xueqi Cheng, Wengang Zhou, and Tie-Yan Liu. 2019. Soft Contextual Data Augmentation for Neural Machine Translation. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Association for Computational Linguistics, 5539–5544.

[21] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 563–575.

[22] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.

[23] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM.

[24] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Phrase-Based & Neural Unsupervised Machine Translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 5039–5049.

[25] Yu Li, Xiao Li, Yating Yang, and Rui Dong. 2020. A Diverse Data Augmentation Strategy for Low-Resource Neural Machine Translation. *Inf.* 11, 5 (2020), 255.

[26] Zhenhao Li and Lucia Specia. 2019. Improving Neural Machine Translation Robustness via Data Augmentation: Beyond Back-Translation. In *Proceedings of the 5th Workshop on Noisy User-generated Text, W-NUT@EMNLP 2019, Hong Kong, China, November 4, 2019*. Association for Computational Linguistics, 328–336.

[27] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.* 18, 2 (2009), 300–336.

[28] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[29] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. IEEE Computer Society, 118–129.

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

[31] Edward Ma. 2019. NLP Augmentation. https://github.com/makcedward/nlpaug.

[32] Vadim Markovtsev and Waren Long. 2018. Public git archive: a big code dataset for all. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 34–37.

[33] George Mathew and Kathryn T. Stolee. 2021. Cross-language code search using static and dynamic analyses. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 205–217. https://doi.org/10.1145/3468264.3468538

[34] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 651–654.

[35] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[36] Anh Tuan Nguyen, Zhaopeng Tu, and Tien N Nguyen. 2016. Do contexts help in phrase-based, statistical source code migration?. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 155–165.

[37] Xuan-Phi Nguyen, Shafiq R. Joty, Kui Wu, and Ai Ti Aw. 2020. Data Diversification: A Simple Strategy For Neural Machine Translation. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[38] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2023. L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models. *arXiv preprint arXiv:2309.17446* (2023).

[39] Pedro Orvalho, Mikolás Janota, and Vasco M. Manquinho. 2022. MultIPAs: applying program transformations to introductory programming assignments for data augmentation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 1657–1661.

[40] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. *CoRR* abs/2308.03109 (2023).

[41] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[42] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020).

[43] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[44] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[45] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving Neural Machine Translation Models with Monolingual Data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

[46] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1609–1620.

[47] Vaibhav, Sumeet Singh, Craig Stewart, and Graham Neubig. 2019. Improving Robustness of Machine Translation with Synthetic Noise. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 1916–1920.

[48] Xinyi Wang, Hieu Pham, Zihang Dai, and Graham Neubig. 2018. SwitchOut: an Efficient Data Augmentation Algorithm for Neural Machine Translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 856–861.

[49] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *J. Syst. Softw.* 190 (2022), 111304.

[50] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[51] Li Zhong and Zilong Wang. 2023. A Study on Robustness and Reliability of Large Language Model Code Generation. *CoRR* abs/2308.10335 (2023).