Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages

Anonymous ACL submission

Abstract

Back-translation is widely known for its effectiveness for neural machine translation when little to no parallel data is available. In this approach, a source-to-target model is coupled with a target-to-source model and trained in parallel. While the target-to-source model generates noisy sources, the source-to-target model is trained to reconstruct the targets and vice versa. Recent developments of multilingual pre-trained sequence-to-sequence models for programming languages have been very effective for a broad spectrum of downstream software engineering tasks. Therefore, it is compelling to train them to build programming language translation systems via back-translation. However, these models cannot be further trained via back-translation since they learn to output sequences in the same language as the inputs during pre-training. As an alternative, we suggest performing back-translation via code summarization and generation. In code summarization, a model learns to generate a natural language (NL) summary given a piece of code, and in code generation, the model learns to do the opposite. Therefore, target-to-source generation in back-translation can be viewed as *target-to-NL-to-source* generation. We take advantage of labeled data for the code summarization task. We show that our proposed framework performs comparably to state-of-the-art methods, if not exceeding their translation performance between Java and Python languages.

1 Introduction

Choice of programming language (PL) in software development depends on the requirement of the software and the available features of a particular PL. In modern API-driven software development, the choice of language often depends on the availability of libraries and APIs. The advent of newer and richer programming languages often requires legacy software to be translated into modernized

Input in Java

```
public static boolean generator(
    PsiBuilder b, int l){
  boolean r = false;
  if(!recursion_guard(b, l)) return r;
  r = generator_0(b, l + 1);
  if(!r) return generator_1(b, l + 1);
}
```

(1) Java to Python Generation

```
public static boolean generator(
    PsiBuilder b, int l){
  boolean r = false;
  if(!recursion_guard(b, l)) return r;
  if(!generator_0(b, l)) return r;
  r = generator_0(b, l + 1);
  if(!r) return generator_1(b, l + 1);
}
```

(2) Java Code to NL Summary

guard is used to determine if a generator is already defined .

(3) NL Summary to Python Code

```
def is_generator(self, name):
  if name in self._generators:
    return True
  if name in self._generators[name]:
    return True
  return False
```

Figure 1: Although PLBART is asked to generate in Python given input in Java (1), it generates in Java (due to its pre-training objective). In contrast, PLBART fine-tuned on code summarization and generation, generates "noisy" translations (as in (2, 3)).

PLs. In theory, modern programming languages' "Turing Completeness" allows rule-based translation of programs from one PL to another. The rule-based translation may require an extensive number of handwritten transformation rules and could end up producing very unreadable source code. In addition, such translation could entail translating the entire library, even if a library implementing similar functionality is available in the target PL.

Aligning libraries and APIs across different PLs is a non-trivial task. Recent progress in Neural Machine Translation (NMT) (Bahdanau et al., 2015;

Vaswani et al., 2017) leveraging pretrained models (Feng et al., 2020a; Guo et al., 2021; Roziere et al., 2021; Ding et al., 2021; Ahmad et al., 2021a; Wang et al., 2021) could be a possible way to learn the alignment between PLs and translate source code across languages.

A significant challenge in supervised learning for NMT is the need for large-scale parallel corpora. For instance, if we are planning to train a translator for `Java` to `Python` translation, we need a considerable number of the same program (*i.e.*, exhibiting the same semantic behavior) in both the languages. Availability of such parallel datasets is a vital challenge in programming language translation (Chen et al., 2018). Back-Translation (BT) (Edunov et al., 2018; Lachaux et al., 2020) is a clever way to learn alignments across different languages. While BT demonstrates success in NMT, those require either (i.) small (perhaps noisy) parallel datasets or (ii.) a model with some capacity of cross-lingual generation - to kickstart the BT-based learning process.

In this research, we investigate the suitability of multilingual Pre-trained Sequence-to-Sequence Model (PSM) (*e.g.*, PLBART (Ahmad et al., 2021a)) for unsupervised programming language translation via BT. In particular, we assume a use case scenario, where there is *no* parallel data available. Without much of a surprise, we empirically found that, while these PSMs are good at generating code in each language, they exhibit very little to no knowledge about the cross-lingual generation since such PSMs are typically trained to reconstruct code sequences from noisy inputs. For example, when we provide the input code in Figure 1 to PLBART and ask to generate Python code without any training, it generates a slight variation of the input Java code, showing its lack of knowledge about cross-lingual generation.

To endow such PSMs with knowledge about cross-lingual generation, we propose the usage of a third language (*i.e.*, English), which is available in bulk quantity. Since a large quantity of monolingual code corpora comes with documentation, which supposedly describes what the source code is doing, we train a Summarize-and-Generate ($\mathcal{S}\&\mathcal{G}$) model that can generate pseudo-parallel code sequences. Figure 1 shows PLBART's behavior when it is further trained via $\mathcal{S}\&\mathcal{G}$. First, given the Java code, it generates a NL summary (figure 1-2), and subsequently generates Python Code (figure 1-3). We empirically show that, even if such $\mathcal{S}\&\mathcal{G}$ model generates noisy parallel sequences, it allows us to employ PSMs in the BT-based training to learn programming language translation.

In summary, we present a Summarize-and-Generate ($\mathcal{S}\&\mathcal{G}$) based approach to enable unsupervised program translation training of PLBART via Back-Translation (BT). Experiment results show that our proposed approach makes PLBART trainable via BT and performs comparably or better than state-of-the-art program translation models.[1]

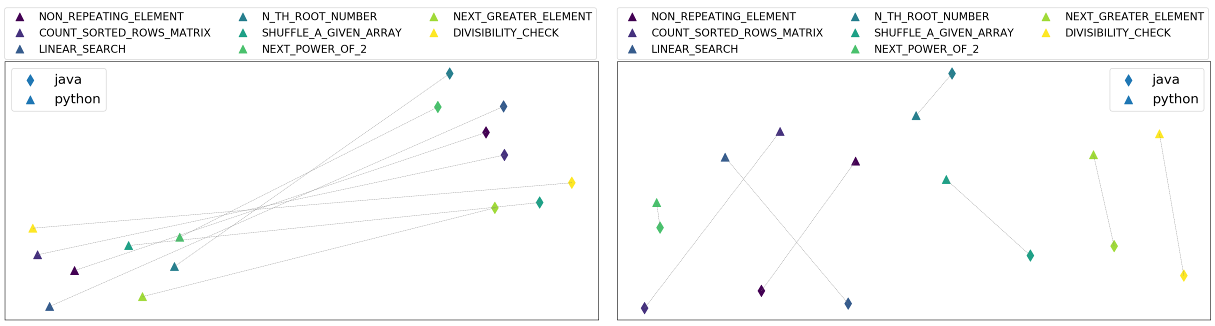
2 Motivation

Recent years saw several Pre-trained Sequence-to-Sequence models (PSM) (Ahmad et al., 2021a; Wang et al., 2021). These models are pre-trained on hundreds of Gigabytes of source code. Thus, we are motivated to investigate their adoption in learning program translation via back-translation in this work. To understand such feasibility, we investigate the program representations generated by the PSM. As a case study, we chose PLBART (Ahmad et al., 2021a) and evaluated its multilingual embeddings as suggested in Artetxe and Schwenk (2019). We find the parallel Java function for each of the 948 Python functions using the parallel dataset proposed in Lachaux et al. (2020). We find the nearest neighbor using cosine similarity between function embeddings and calculate the error rate. Unsurprisingly, PLBART performs poorly in function retrieval with an 87.5% error rate.

In comparison, we fine-tune PLBART jointly on code summarization and generation in Java and Python. Repeating the experiment of function retrieval, we find fine-tuned PLBART's error rate drops to *23.7%*. To visually illustrate the embeddings produced by PLBART and its fine-tuned variant, we provide a T-SNE plot of 8 sample functions' embedding in Figure 2. We see the functions that belong to the same language are clustered together while the same functions in two different languages are far apart from each other (see Figure 2a).

In contrast, the fine-tuned PLBART breaks up the intra-language clusters and brings functions in different languages close to each other in the embedding space (see Figure 2b). These results motivate us to initialize the translation models with fine-tuned PLBART on code summarization and generation for back-translation as it learned some alignment across programming languages.

[1]We have made our code publicly available at `https://github.com/hidden/hidden`.



(a) PLBART (b) PLBART + $\mathcal{S}\&\mathcal{G}$

Figure 2: T-SNE plot of function embeddings of Java and Python functions. Figure 2a shows the embedding generated by PLBART model. Figure 2b are the generated embedding when the PLBART is finetuned to jointly summarize code to NL and generate code from NL (PLBART + $\mathcal{S}\&\mathcal{G}$). While PLBART clusters programs from each individual PLs, same program in different PLs are brought closer to each other by PLBART + $\mathcal{S}\&\mathcal{G}$.

3 Approach

Sequence-to-sequence models, such as PLBART (Ahmad et al., 2021a), CodeT5 (Wang et al., 2021), map source code sequences into a shared multilingual space by pre-training on multiple programming languages jointly using unlabeled data (*e.g.*, source code from Github). The pre-training objective of these models is either denoising autoencoding (DAE) or fill-in-the-blank, where the models reconstruct the original code snippet or predict the missing code tokens given a corrupted code snippet. Although pre-trained jointly on many languages, these models only learn to generate in the same language as input. As a result, these models are not trainable via back-translation (BT) to learn programming language translation in an unsupervised fashion. As an alternative, we propose translating to and from natural language to perform back-translation between two programming languages. We refer to translating to and from natural language as code summarization and code generation, respectively. Our proposal is motivated based on the availability of *bimodal* data, source code, and their summaries that are used to train code summarization and generation models.

3.1 Back-translation

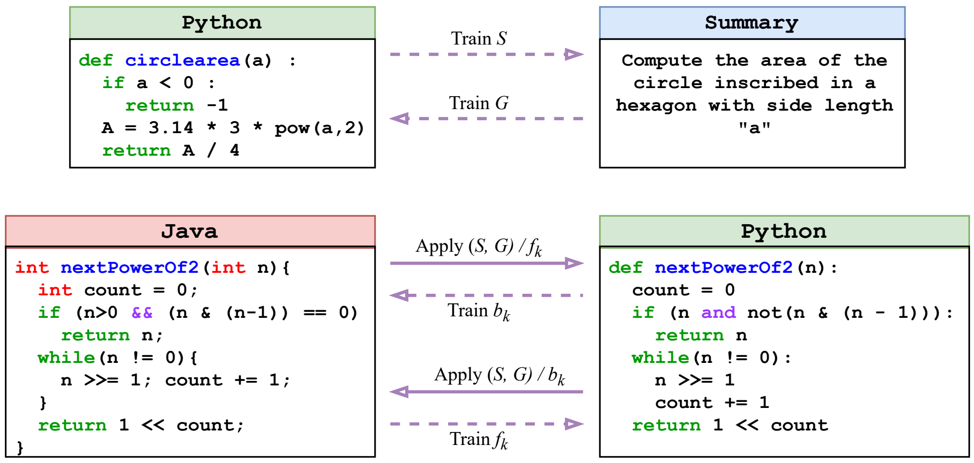
Back-translation (BT) is one of the most popular ways for unsupervised machine translation (Artetxe et al., 2018b; Lample et al., 2018a,b). In this approach, we leverage monolingual data in an unsupervised fashion. BT jointly trains a source-to-target model coupled with a backward target-to-source model. The target-to-source model translates target sequences into the source language, producing noisy sources corresponding to the ground truth target sequences. The source-to-target model is then trained to generate the targets from the noisy sources and vice versa. The two models are trained in parallel until convergence. This training procedure is widely known as *online back-translation* and the focus of this work.

Back-translation uses a target-to-source model to generate noisy sources and trains a source-to-target model to reconstruct the targets. Specifically, in each step k (a mini-batch update), back-translation performs the following:

$$\begin{aligned}
\mathcal{P}_k^{(f)} &= \{(x, f_{k-1}(x)) | x \in \mathcal{D}_{\text{source}}\} \\
b_k &= TRAIN^{\text{target}\to\text{source}}\left(\mathcal{P}_k^{(f)}\right) \\
\mathcal{P}_k^{(b)} &= \{(b_k(y), y) \,| y \in \mathcal{D}_{\text{target}}\} \\
f_k &= TRAIN^{\text{source}\to\text{target}}\left(\mathcal{P}_k^{(b)}\right).
\end{aligned} \tag{1}$$

Here, $\mathcal{D}_{source}, \mathcal{D}_{target}$ represents unlabeled data in source and target languages and $TRAIN$ indicates standard sequence-to-sequence training.

Generally, the training via back-translation starts from a forward (f_0) and a backward (b_0) model that is trained using parallel data (small gold-standard or large-scale but noisy). Then an extensive collection of unlabeled data is used to train the translation models. In this work, we assume there is *no* parallel data available across programming languages. We initialize the forward and backward model with the pre-trained language model, PLBART. As mentioned before, PLBART cannot generate code in a language different from the input (not even a noisy code) (for example, figure 1-1). Therefore, we propose jointly fine-tuning PLBART on code summarization and generation on multiple programming



```python
def circlearea(a) :
  if a < 0 :
    return -1
  A = 3.14 * 3 * pow(a,2)
  return A / 4
```

```
Compute the area of the
circle inscribed in a
hexagon with side length
"a"
```

```java
int nextPowerOf2(int n){
  int count = 0;
  if (n>0 && (n & (n-1)) == 0)
    return n;
  while(n != 0){
    n >>= 1; count += 1;
  }
  return 1 << count;
}
```

```python
def nextPowerOf2(n):
  count = 0
  if (n and not(n & (n - 1))):
    return n
  while(n != 0):
    n >>= 1
    count += 1
  return 1 << count
```

Figure 3: Overview of our proposed back-translation framework to train PLBART. In the first $k = m$ steps (out of total N training steps), we use a multilingual code summarization and generation model $(\mathcal{S}, \mathcal{G})$ to perform back-translation. In the remaining steps $(N - m)$, PLBART is trained via standard back-translation method.

languages in a supervised setting. Then use the resulting model to initialize the forward and backward model (f_0, b_0) for back-translation.

3.2 Code Summarization and Generation

Source code documentation (*e.g.,* docstring or comment) written by software developers are available along with source code on a large scale. Such documentation has been the key source to form source code summarization datasets (Wan et al., 2018; Hu et al., 2018; LeClair and McMillan, 2019; Husain et al., 2019). These datasets are also utilized in natural language (NL) to code generation studies (Parvez et al., 2021). It is tangible that we can use a code summarization and generation model to translate programming languages. Such a model would first generate an NL summary from code in the source language and then generate code in the target language from the previously generated NL summary. As we show in the evaluation, such an approach does not work well in practice (see table 2); however, code summarization and generation models are viable proxies to generate noisy translations. This enables us to train PLBART, to begin with generating noisy translations and further learn to improve in a self-supervised fashion when trained via back-translation. Formally, we jointly train PLBART in a supervised setting to learn code summarization ($\mathcal{S}$) and generation ($\mathcal{G}$):

$$\begin{aligned} \mathcal{S} &= TRAIN^{\text{Code}\rightarrow\text{Summary}}\left(\mathcal{P}_{c,s}\right) \\ \mathcal{G} &= TRAIN^{\text{Summary}\rightarrow\text{Code}}\left(\mathcal{P}_{c,s}\right) \end{aligned} \tag{2}$$

where $\mathcal{P}_{c,s}$ is estimated using the code-to-text benchmark from CodeXGlue (Lu et al., 2021). We follow Tang et al. (2021) to perform multilingual fine-tuning of PLBART (in Java and Python) to learn $\mathcal{S}$ and $\mathcal{G}$.

3.3 Summarize–Generate to Back-translate

The back-translation method requires the *target-to-source* (b) and *source-to-target* (f) models to generate semantically equivalent (parallel) sequences in the source and target languages, respectively. These parallel sequences are then used to train the models to learn translations as described in Eq. (1). Specifically, the parallel sequences $\{(\hat{x}, y), (x, \hat{y})\}$ created by computing $\hat{y} \leftarrow f_{k-1}(x)$ and $\hat{x} \leftarrow b_k(y)$ kick-start the learning process in back-translation. Generally, the curated parallel sequences tend to be noisy since we do not have access to accurate target-to-source and source-to-target models. However, both the models trained in parallel via back-translation until convergence, resulting in useful translation models.

The advancements of pre-trained sequence-to-sequence models on programming languages enables us to use them in initializing the source-to-target (f) and target-to-source (b) models for back-translation. Presumably, such pre-trained models should facilitate the learning process during training. Yet, their pre-training objective – *i.e.,* reconstruction of original input from a noisy source limits their ability to generate code snippets across languages (as shown in Figure 1). For example, PLBART as $f(\cdot)$ and $b(\cdot)$ would reconstruct the input, resulting in $f_{k-1}(x) \approx x$ and $b_k(y) \approx y$. As a result, the models will learn to merely copy the input sequences rather than translate them.

To this end, we propose to make use of available parallel data between programming and natural lan-

Algorithm 1 Training Procedure

```
Input: Monolingual data $\mathcal{D}_s$ and $\mathcal{D}_t$ of languages $s$ and $t$; number of initial steps $m$; number of total steps $I$; code summarizer $\mathcal{S}(\cdot,\cdot)$; code generator $\mathcal{G}(\cdot,\cdot)$; parameters $\theta$ to initialize the forward and backward translation models $f(\cdot,\cdot)$ and $b(\cdot,\cdot)$.
Output: Final model parameters $\theta$.
1: for $k = 0, \cdots, I$ do
2:     $y \leftarrow (y_s \sim \mathcal{D}_s) \cup (y_t \sim \mathcal{D}_t)$
3:     if $k \leq m$ then
4:         $x_{nl} \sim \mathcal{S}(\cdot|y)$            ▷ code-to-summary
5:         $\hat{x} \sim \mathcal{G}(\cdot|x_{nl})$        ▷ summary-to-code
6:     else
7:         $\hat{x} \leftarrow (x_s \sim b(\cdot|y_t)) \cup (x_t \sim f(\cdot|y_s))$
8:     Update $\theta$ by maximizing log-likelihood of $f(\hat{x}_s, y_t)$ and $b(\hat{x}_t, y_s)$
```

guages to fine-tune PLBART and then use its parameters to initialize source-to-target (f) and target-to-source (b) models for back-translation. Consequently, we revise the back-translation training method outlined in Eq. (1) to follow a *two-step generation* process to perform back-translation: code-to-summary generation in natural language followed by summary-to-code generation in the source language. Formally, the first m steps ($k \leq m$) of back-translation is performed as:

$$\begin{aligned}\mathcal{P}_k^{(f)} &= \{(x, \mathcal{G}(\mathcal{S}(x))) \,|x \in \mathcal{D}_{\text{source}}\} \\ \mathcal{P}_k^{(b)} &= \{(\mathcal{G}(\mathcal{S}(y)), y) \,|y \in \mathcal{D}_{\text{target}}\}.\end{aligned} \tag{3}$$

We find the noisy parallel sequences[2] generated via summarization and generation commences the learning process. The overall idea of our proposed framework is illustrated in Figure 3 and the Algorithm 1 describes the training procedure. Note that we find it is sufficient to apply our proposed summarization-generation based back-translation only for the first m steps as the source-to-target and target-to-source models gradually learn to translate, the standard back-translation training reinstated.

4 Experiment Setup

4.1 Models and Baselines

Our model As noted earlier, PLBART (Ahmad et al., 2021a) and CodeT5 (Wang et al., 2021) are two popular sequence-to-sequence models pre-trained on source code that cannot be trained via back-translation (BT). As an alternative, our proposed approach can be leveraged to train both models to learn programming language translation in an unsupervised fashion. In this work, we chose PLBART to perform experiments and show the effectiveness of our proposed approach.

Baseline Models

We compare our proposed approach applied to PLBART with the following existing approaches.

• **j2py** is a framework that translates Java source code to Python.[3] It follows handwritten rules manually built using expert knowledge.

• **Summarize-and-Generate** ($\mathcal{S}\&\mathcal{G}$) performs code-to-code translation via two steps, code-to-summary and summary-to-code generation. We evaluate the $\mathcal{S}\&\mathcal{G}$ model (as in Eq. (2)) that is used to perform code summarization and generation in our proposed framework to train PLBART via BT.

• **TransCoder** is a neural translation model for programming languages (Lachaux et al., 2020). TransCoder is developed by pretraining Transformer (Vaswani et al., 2017) via masked language modeling (MLM) objective (Devlin et al., 2019) on monolingual source code datasets. In a second step, TransCoder is trained via denoising autoencoding (DAE) and BT. In this work, we consider TransCoder as the *primary* baseline.

• **DOBF** Roziere et al. (2021) proposed a pretraining objective, DOBF, that leverages the structural aspects of programming languages. According to this pretraining paradigm, the identifiers (class, function, and variable names) in code snippets are obfuscated, and a model is trained to recover the original names. DOBF shares the exact same neural architecture as TransCoder.

4.2 Evaluation Dataset and Metrics

Evaluation Dataset Lachaux et al. (2020) proposed an evaluation dataset composed of parallel functions in Java, Python, and C++ languages. The dataset consists of 464 Java to Python and 482 Python to Java test examples, where each example is accompanied by 10 unit test cases.

Evaluation Metrics

• **BLEU** measures n-gram overlap between a generated translation and a collection of reference translations (Papineni et al., 2002).

[2]The output sequences are still noisy since the code summarization and generation models are not highly accurate although trained in a supervised fashion.

[3]https://github.com/natural/java2python

	Java	Python
Github - *unimodal* data		
Nb of functions	44.5 M	42.0 M
Nb of tokens	3.3 B	4.1 B
CodeNet - *unimodal* data		
Nb of functions	0.42 M	0.15 M
Nb of tokens	47.3 M	17.0 M
CodeXGlue - *bimodal* data		
Nb of functions	164,923	251,818
Nb of tokens	21.2 M	44.3 M

Table 1: Statistics of the data used to train PLBART at different stages in this work. *Bimodal* data refers to parallel function-summary pairs, while *unimodal* data refers to monolingual (and unparallel) functions.

• **Exact Match (EM)** represents the percentage of generated translations exactly match with the collection of reference translations.

• **CodeBLEU** measures grammatical and logical correctness in addition to n-gram overlap between generated and reference translations (Ren et al., 2020). CodeBLEU assesses grammatical and logical correctness based on the abstract syntax tree and the data-flow structure.

• **Computational Accuracy (CA)** evaluates if a generated function outputs the same as the reference when given the same set of inputs. Lachaux et al. (2020) introduced this evaluation metric to perform evaluation based on unit tests.

4.3 Training Datasets and Preprocessing

Code Summarization and Generation Lu et al. (2021) curated a code summarization dataset consisting of code and summary pairs based on the CodeSearchNet dataset (Husain et al., 2019). We use this dataset in Java and Python programming languages to train the code-to-summary and summary-to-code generation models.

Back-translation (BT) For BT training (as discussed in § 3.3), we use the GitHub public dataset available on Google BigQuery (Hoffa, 2016).[4] Note that the Github dataset is composed of source code that covers a wide variety of programming topics (as they come from various projects). In contrast, the evaluation dataset is composed of programming problems covering basic data structure and algorithmic concepts. Therefore, to investigate the impact of data on BT training, we alternatively chose *unparallel* code samples in Java and Python from CodeNet (Puri et al., 2021). The CodeNet dataset is collected from two online judge websites, *AIZU Online Judge* and *AtCoder*, and composed of submissions for 4053 problems. We use the deduplicate accepted solutions of the problems for BT training. Presumably, CodeNet and the evaluation dataset have a similar nature that should positively impact downstream translation performance.

Preprocessing We use `tree_sitter`[5] for tokenizing Java functions and the tokenizer of the standard library for Python.[6] We extract standalone functions[7] from the BT training datasets following the function extraction technique from Lachaux et al. (2020). We filter the standalone functions exceeding a maximum length of 256 to cope with our computational resources. The statistics of the preprocessed datasets are presented in Table 1.

4.4 Implementation Details

We jointly train PLBART on code summarization and generation in Java and Python using the authors' provided code.[8] Subsequently, we further train PLBART via back-translation as described in Algorithm 1. We set $I = 10,000$ and tuned $m = 200$.[9] We train PLBART using 8 Nvidia GeForce RTX 2080 Ti GPUs, and the effective batch size is maintained at 1024 instances at both training stages. We optimize PLBART with the Adam optimizer (Kingma and Ba, 2015), a learning rate of 10e-4, and use a polynomial learning rate decay scheduling. The best models are selected based on the validation BLEU scores. We implement our approach in Fairseq (Ott et al., 2019) and use float16 operations to speed up training.

Decoding During inference, we use beam search decoding (Koen, 2004) to generate multiple translations using PLBART. We chose greedy search (Beam 1) as the default decoding scheme for validation and evaluation. However, following Lachaux et al. (2020), we report two sets of results for the computational accuracy (CA) metric: CA@n B=n,

[4]https://console.cloud.google.com/marketplace/product/github/github-repos

[5]https://github.com/tree-sitter/py-tree-sitter

[6]https://docs.python.org/3/library/tokenize.html

[7]Standalone functions can be used without instantiating a class. In Java, this corresponds to static methods, and in Python, it corresponds to functions outside classes.

[8]https://github.com/wasiahmad/PLBART/tree/main/multilingual

[9]We tuned m in the range [100, 1000] with 100 steps.

Models	Java → Python				Python → Java			
	BLEU	EM	CodeBLEU	CA	BLEU	EM	CodeBLEU	CA
j2py*	-	-	-	38.3	-	-	-	-
TransCoder*	68.1	3.7	-	35.0	64.6	0.8	-	24.7
TransCoder w/ DOBF*	-	-	-	**49.2**	-	-	-	39.5
$\mathcal{S}\&\mathcal{G}$ (2)	7.6	0.0	15.8	0.2	12.4	0	16.3	0.2
PLBART **(this work)**								
trained via BT	31.2	0.0	36.6	0.0	31.7	0.0	32.1	0.0
trained via BT (via $\mathcal{S}\&\mathcal{G}$)	63.2	2.5	60.4	35.4	64.1	1.2	65.9	**43.8**

Table 2: Evaluation results of the baselines models and our proposed framework using greedy decoding. * indicates scores reported from Lachaux et al. (2020).

Models	TransCoder	PLBART
Java → Python		
CA@1 B=1	35.0	35.5
CA@1 B=10	49.0	38.4
CA@5 B=5	60.0	47.0
CA@10 B=10	64.4	50.0
Python → Java		
CA@1 B=1	24.7	43.8
CA@1 B=10	36.6	45.4
CA@5 B=5	44.3	52.5
CA@10 B=10	51.1	55.0

Table 3: Computational accuracy (CA@m) with beam search decoding and comparison between TransCoder and PLBART. TransCoder's performances are reported from Lachaux et al. (2020). The value B indicates the beam size. CA@m B=n means that we use beam decoding to generate n translations, and select the top m translations based on their log-probability scores.

the percentage of functions with at least one correct translation in the beam (of size n), and CA@1 B=n the percentage of functions where the hypothesis in the beam with the highest log-probability is a correct translation.

5 Results

5.1 Main Result

Table 2 shows the performance of our proposed approach and the baseline models on both Java to Python and Python to Java translation. We begin by comparing PLBART directly used in back-translation (BT) training vs. our proposed approach (last block in Table 2). As emphasized before, PLBART does not know to generate across languages, so when the model is trained via BT, it only learns to copy the input sources. As a result, PLBART scores 0% EM and 0% CA, while 30+ BLEU and CodeBLEU scores indicate that they are not a reliable indicator of translation correctness.

In contrast, following our proposed approach of summarizing and generating to back-translate, PLBART trained via BT (via $\mathcal{S}\&\mathcal{G}$) achieves comparable or better performance than TransCoder.[10] We further compare them using beam search decoding in Table 3. This work studies the feasibility of using pre-trained sequence-to-sequence models for unsupervised programming language translation via BT. The experimental results conclude that such models cannot directly be used in BT training; however, training via $\mathcal{S}\&\mathcal{G}$ empowers them to generate across languages and further be trained via BT to learn to translate.

5.2 Analysis

Summarize and generate to create parallel data Our proposed approach generates parallel code sequences on the fly (online) for training. An alternative to our approach is to use a code summarization and generation model to create parallel code sequences (offline) and warm-start PLBART for back-translation-based training. We compare these two approaches in Table 4, and the results show that both approaches perform comparably. However, it is essential to note that the online setting gives us flexibility as we can tune the number of initial steps (m in Algorithm 1). In contrast, the offline setting requires generating a sufficiently large number of parallel code sequences.

Impact of in-domain training data The evaluation dataset comprises solutions to programming problems involving data structures and algorithm concepts. While Github offers large-scale unlabeled data, most of its code belongs to software projects that use APIs and advanced functionalities. Therefore, we utilize an alternative dataset called

[10]Note that, while comparing PLBART with TransCoder on the translation performance, their differences (shown in Table 9) should be taken into account.

Init. Checkpoint	Java to Python				Python to Java			
	BLEU	EM	CodeBLEU	CA	BLEU	EM	CodeBLEU	CA
Warm-start w/ PD	59.5	2.5	57.1	**37.9**	62.6	**1.7**	65.9	43.8
Proposed approach	**63.2**	2.5	**60.4**	35.4	**64.1**	1.2	65.9	43.8

Table 4: Comparison between PLBART warm-started using parallel data (PD) and our approach to summarize and generate to back-translate on the fly during the initial steps of back-translation training.

Data Source	Java to Python				Python to Java			
	BLEU	EM	CodeBLEU	CA	BLEU	EM	CodeBLEU	CA
Github	63.2	**2.5**	60.4	35.4	64.1	**1.2**	65.9	43.8
CodeNet	**65.6**	1.6	**61.7**	**50.9**	**65.1**	**1.2**	**68.5**	**46.5**

Table 5: PLBART evaluation results when our proposed framework uses data from Github (available via BigQuery (Hoffa, 2016)) and competitive programming sites (available via CodeNet (Puri et al., 2021)).

CodeNet collected from two online judge websites. We refer to this dataset as in-domain data, and we compare its usage with Github data on BT-based training. The results in Table 5 show that the use of in-domain data significantly boosts the translation performance, notably in Java-to-Python translation. A detailed error analysis reveals that such performance boost is due to reduction in `TypeError`. Due to the page limit, we present the findings of the error analysis in the Appendix.

In addition, we discuss the limitations and risks of using our proposed model in the Appendix.

6 Related Work

Programming Language Translation Translating programs or source code across different programming languages (PL) requires a profound understanding of the PLs. Having strictly defined syntax and semantics, PLs are suitable for phrase-based statistical machine translation (Nguyen et al., 2013; Karaivanov et al., 2014; Aggarwal et al., 2015). Chen et al. (2018) introduced a tree to tree machine translation to translate programs and to learn the syntactic alignment between source and target PL. Recently proposed pretrained programming language models showed promising results in translating programs across PLs (Feng et al., 2020b; Guo et al., 2021; Ahmad et al., 2021a,b). However, these approaches require a set of parallel programs to train the encoder-decoder model.

Recently proposed Transcoder (Lachaux et al., 2020) shows initial success results in unsupervised program translation, eliminating the requirement of bi-modal data. They achieve such jointly training a model using XLM (Conneau and Lample, 2019), Denoising Auto Encoding (DAE) (Vincent et al., 2008), and Back-Translation(BT) (Lample et al., 2018a). This work empirically investigated the suitability of adopting BT to train existing pre-trained encoder-decoder models and proposed an alternative via summarization and generation.

Unsupervised Machine Translation via Back-translation Gathering sufficiently large parallel corpora has been a major challenge for Machine Translation (MT) (Guzmán et al., 2019). Several research efforts are invested in learning MT using monolingual data (Artetxe et al., 2018a,b; Lachaux et al., 2020) to solve this problem. For example, Gulcehre et al. (2015) proposed integration of a Language model integrated into the decoder. He et al. (2016) proposed Neural MT (NMT) as a bidirectional and dual learning task. More recent advancements in unsupervised MT leverages back-translation (BT) (Sennrich et al., 2016; Lample et al., 2018a,b). In back-translation, the target-to-source model generates noisy sources given target sequences and then trains the source-to-target model to reconstruct the targets and vice versa. While BT has been widely adopted for unsupervised NMT, it is used in other applications (Zhu et al., 2017; Hoffman et al., 2018; Shen et al., 2017; Yang et al., 2018; Zhang et al., 2019).

7 Conclusion

In this research, we show that pretrained sequence-to-sequence models (*e.g.*, PLBART) are not suitable for direct adaptation via back-translation to learn to translate. To address the issue, we propose to use code summarization and generation as an alternative to performing back-translation. We show that our proposed approach turns PLBART into a translation model that performs comparably to existing unsupervised translation models.

References

Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021b. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2018a. Unsupervised statistical machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3632–3642, Brussels, Belgium. Association for Computational Linguistics.

Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. 2018b. Unsupervised neural machine translation. In *International Conference on Learning Representations*.

Mikel Artetxe and Holger Schwenk. 2019. Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond. *Transactions of the Association for Computational Linguistics*, 7:597–610.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31*, pages 2547–2557. Curran Associates, Inc.

Alexis Conneau and Guillaume Lample. 2019. Cross-lingual language model pretraining. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 7059–7069. Curran Associates, Inc.

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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Contrastive learning for source code with structural and functional properties. *arXiv preprint arXiv:2110.03868*.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 489–500, Brussels, Belgium. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020b. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2015. On using monolingual corpora in neural machine translation. *arXiv preprint arXiv:1503.03535*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Francisco Guzmán, Peng-Jen Chen, Myle Ott, Juan Pino, Guillaume Lample, Philipp Koehn, Vishrav Chaudhary, and Marc'Aurelio Ranzato. 2019. The FLORES evaluation datasets for low-resource machine translation: Nepali–English and Sinhala–English. 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)*, pages 6098–6111, Hong Kong, China. Association for Computational Linguistics.

Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

Felipe Hoffa. 2016. Github on bigquery: Analyze all the open source code.

Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei Efros, and Trevor Darrell. 2018. CyCADA: Cycle-consistent adversarial domain adaptation. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1989–1998. PMLR.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Philipp Koen. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas: Technical Papers*, pages 115–124, Washington, USA. Springer.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018a. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations*.

Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018b. Phrase-based & neural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5039–5049, Brussels, Belgium. Association for Computational Linguistics.

Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, Minneapolis, Minnesota. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

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*, pages 651–654.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. In *Advances in Neural Information Processing Systems*.

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 (Volume 1: Long Papers), pages 86–96, Berlin, Germany. Association for Computational Linguistics.

Tianxiao Shen, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2017. Style transfer from non-parallel text by cross-alignment. In *Advances in Neural Information Processing Systems 30*.

Yuqing Tang, Chau Tran, Xian Li, Peng-Jen Chen, Naman Goyal, Vishrav Chaudhary, Jiatao Gu, and Angela Fan. 2021. Multilingual translation from denoising pre-training. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3450–3466, Online. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 397–407, New York, NY, USA. Association for Computing Machinery.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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. Association for Computational Linguistics.

Zichao Yang, Zhiting Hu, Chris Dyer, Eric P Xing, and Taylor Berg-Kirkpatrick. 2018. Unsupervised text style transfer using language models as discriminators. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7298–7309.

Zhirui Zhang, Shuo Ren, Shujie Liu, Jianyong Wang, Peng Chen, Mu Li, Ming Zhou, and Enhong Chen. 2019. Style transfer as unsupervised machine translation. In *Thirty-Third AAAI Conference on Artificial Intelligence*.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232.

Supplementary Material: Appendices

	TransCoder	PLBART
Java → Python		
#Tests	464	464
Error	149	146
Failure	93	146
Success	218	164
EM	17	24
Timeout	4	8
Python → Java		
#Tests	482	482
Error	201	212
Failure	118	53
Success	157	211
EM	6	2
Timeout	6	6

Table 6: Detailed results of computational accuracy using greedy decoding for Java ↔ Python translation.

A Analysis of Computational Accuracy

Table 6 shows the breakdown of computational accuracies for Java-to-Python and Python-to-Java translation for TransCoder and our proposed approach using PLBART. We execute the generated function and match the output *w.r.t.* the expected output. TransCoder results in 149 error cases, 93 failure cases, and 218 success cases in Java-to-Python translation, with 17 solutions matching the ground truth. In contrast, PLBART results in 146 error cases, 146 failure cases, 164 success cases. Out of these 164 successes in PLBART, 24 solutions exactly match the target solution.

For Python-Java translation, TransCoder results in 201 errors, 118 failures, and 157 successes, out of which 6 are an exact match. On the other hand, in the case of PLBART, there are 212 error cases, 53 failure cases, and 211 success cases, out of which 2 exactly match the target solution.

B Error Analysis

We further analyze the error cases for TransCoder and our proposed approach using PLBART. Since Python is an interpreted language, syntactic and semantic errors are caught at runtime. Thus, we categorize all errors for Java-to-Python translation as runtime errors. Table 7 shows the errors in both Java-to-Python and Python-to-Java translation. While PLBART is susceptible to `TypeError`, TransCoder is disproportionately susceptible to `SyntaxError`. In the case of Python-to-Java translation, PLBART exhibits more Compilation errors, but TransCoder exhibits more Runtime errors. The most common type of compilation errors in both TransCoder and PLBART is `TypeError`. The most common runtime error in Python-to-Java translation is `InderOutOfBoundException` for both models, where TransCoder exhibits more than twice the number of such errors in PLBART.

Finally, we identified the top five error categories (which accounts for 123 errors out of 146) exhibited by PLBART in Java-to-Python translation and analyzed the error messages. In most cases, `TypeError` and `ValueError` are due to mismatch in underlying data type of variable. Table 8 shows the detailed statistics of different error types, sub-types, and their frequencies.

Error Category	TransCoder	PLBART
#Errors (Java → Python)	149	146
Compilation	-	-
Runtime	149	146
TypeError	47	61
IndexError	18	20
NameError	17	16
ValueError	11	15
UnboundLocalError	13	11
Others	17	14
SyntaxError	26	9
#Errors (Python → Java)	201	212
Compilation	151	180
TypeError	89	108
CantFindSymbol	23	30
SyntaxError	14	25
BadOperand	15	12
Others	10	5
Runtime	50	27
IndexOutOfBoundsE.	40	15
NumberFormatE.	5	6
NullPointerE.	2	3
Others	3	3

Table 7: Category of errors made by the TransCoder and PLBART translation models. The error categories are sorted based on the PLBART's error count on the respective category. In Python → Java runtime error categories, "E." stands for "Exception".

Error Category	Count
Type Error	61
list indices must be integers or slices, not **A**	18
A object does not support item assignment	13
A object cannot be interpreted as an integer	8
unsupported/bad operand type(s)	10
int object is not iterable/callable/subscriptable	6
Others	6
Index Error	20
B index out of range	19
others	1
Name Error	16
name **C** is not defined	16
Value Error	15
not enough values to unpack	7
too many values to unpack	3
the truth value of an array with more than one element is ambiguous	3
others	2
Unbound Local Error	11
local variable **D** referenced before assignment	11

Table 8: Analyzing the five most frequent error cases (123 out of 146) encountered in PLBART generated Java to Python translation. **A** and **B** indicate {bool, int, tuple, str, range} and {string, list}, respectively. **C** and **D** indicate identifier (class, function, variable) names.

C Qualitative Examples

Figure 4 shows an example of Java-to-Python translation by PLBART. The translated code is both syntactically and semantically correct *i.e.,* our compiler could successfully parse and build the translated code. It passed 2 test cases out of 10 when executed. The translated code is slightly different from the input Java code. In particular, line 13 in the input Java code is a loop that iterates backward (decreasing order). However, line 12 in the generated python code iterates forward (increasing order). If the generated python code was `range(c-1, 0, -1)` instead of `range(c-1)`, it would pass all the test cases. We attribute such behavior to the fact that `range(*)` is much more frequent pattern than `range(*, 0, -1)` in python code.

D TransCoder vs. PLBART

As we consider TransCoder as the primary baseline of our proposed approach using PLBART, for the sake of fairness, we compare them in terms of model structure and training setting. Table 9 presents the comparison. TransCoder and PLBART both use the Transformer (Vaswani et al., 2017) architecture, but TransCoder is a twice as large model as PLBART. Both the models have gone through a two-stage training process. In Stage-1, TransCoder is pre-trained via MLM using 920B tokens, while PLBART is pre-trained via DAE using 87B tokens. In Stage-2, TransCoder leverages 625M tokens and jointly trained via DAE and BT. In comparison, PLBART is trained via BT using 430M tokens.

Why TransCoder does not suffer from the same language generation issue? In Stage-1 pre-training, TransCoder only trains the Transformer Encoder and then initializes a decoder with Encoders' parameters, and the cross attention sub-layers are randomly initialized. We speculate that such random initialization leaves TransCoder unbiased towards generating in the same language as input. Moreover, PLBART uses language ID token as a prefix to generate in the target language. We noticed that PLBART's decoder disregards the prefix token if not fine-tuned to generate in the target language. On the other hand, TransCoder uses language embeddings with each token in the input. Intuitively, this does not allow the TransCoder's decoder to ignore the language

Input function in Java

```
1  static int sortedCount(int mat[][],
2        int r, int c){
3    int result = 0;
4    for(int i = 0; i < r; i++){
5      int j;
6      for(j = 0; j < c - 1; j++)
7        if(mat[i][j + 1] <= mat[i][j])
8          break;
9      if(j == c - 1) result++;
10   }
11   for(int i = 0; i < r; i++){
12     int j;
13     for(j = c - 1; j > 0; j--)
14       if(mat[i][j - 1] <= mat[i][j])
15         break;
16     if(c > 1 && j == 0) result++;
17   }
18   return result;
19 }
```

Translated function in Python

```
1  def sorted_count(mat, r, c):
2    result = 0
3    for i in range(r):
4      j = 0
5      for j in range(c - 1):
6        if mat[i][j + 1] <= mat[i][j]:
7          break
8      if j == c - 1:
9        result += 1
10   for i in range(r):
11     j = c - 1
12     for j in range(c - 1):
13       if mat[i][j - 1] <= mat[i][j]:
14         break
15     if c > 1 and j == 0:
16       result += 1
17   return result
```

Figure 4: An example of Java to Python translation by PLBART that passes **2 out of 10** unit test cases. Line no. 13 (marked in green) in the Java function is incorrectly translated in python (line no. 12, marked in red). Replacing the *range* function parameter "(c-1)" by "(c - 1, 0, -1)" would make the translated function pass all the test cases.

	TransCoder	PLBART
#layers (encoder)	6	6
#layers (decoder)	6	6
#heads	8	12
Model dim	1024	768
Vocab size	64,000	50,000
Total parameters	312 M	140 M
Stage1: Pre-training		
Objective	MLM	DAE
Total tokens	920 B	87 B
Token types	BPE	Sentencepiece
Languages	Java, Python, C++	Java, Python, English
Stage2: Training		
Objective	DAE+BT	BT
Total tokens	625 M	430 M
Token types	BPE	Sentencepiece
Languages	Java, Python, C++	Java, Python

Table 9: TransCoder vs. PLBART.

information. For example, with position index "0" and language ID "Python", TransCoder is more likely to generate "def" token and less likely to generate "static" or "int" since they do not appear in the Python language. In essence, unlike PLBART, TransCoder does not suffer from the issue of sequence-to-sequence models being unable to generate across languages.

E Limitations and Risks

One of the risks of using our developed translation model is we used the Github dataset for training that may contain information that uniquely identifies an individual or offensive content. Since we are developing the translation model for research purposes *only*, we believe our usage of the Github data does not violate their licensing terms and conditions. While we do not present it as a justification, the PLBART model was pre-trained on the Github data that may include sensitive information. As our intention is to develop a programming language translation model, it is unlikely to generate sensitive information unless it is provided such information as input.