Unified Pre-training for Program Understanding and Generation

Wasi Uddin Ahmad[§*], **Saikat Chakraborty**[†*], **Baishakhi Ray**[†], **Kai-Wei Chang**[§]
[§]University of California, Los Angeles, [†]Columbia University
[§]{wasiahmad, kwchang}@cs.ucla.edu, [†]{saikatc, rayb}@cs.columbia.edu

Abstract

Code summarization and generation empower conversion between programming language (PL) and natural language (NL), while code translation avails the migration of legacy code from one PL to another. This paper introduces PLBART, a sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks. PLBART is pre-trained on an extensive collection of Java and Python functions and associated NL text via denoising autoencoding. Experiments on code summarization in the English language, code generation, and code translation in seven programming languages show that PLBART outperforms or rivals state-of-the-art models. Moreover, experiments on discriminative tasks, *e.g.,* program repair, clone detection, and vulnerable code detection, demonstrate PLBART's effectiveness in program understanding. Furthermore, analysis reveals that PLBART learns program syntax, style (*e.g.,* identifier naming convention), logical flow (*e.g.,* `if` block inside an `else` block is equivalent to `else if` block) that are crucial to program semantics and thus excels even with limited annotations.

1 Introduction

Engineers and developers write software programs in a programming language (PL) like Java, Python, etc., and often use natural language (NL) to communicate with each other. Use of NL in software engineering ranges from writing documentation, commit messages, bug reports to seeking help in different forums (*e.g.,* Stack Overflow), etc. Automating different software engineering applications, such as source code summarization, generation, and translation, heavily rely on the understanding of PL and NL—we collectively refer them as PLUG (stands for, Program and Language Understanding and Generation) applications or tasks.

Program snippet in Python

```python
def sort_list(uns):
    return sorted(uns, key=lambda x:x[0])
```

Program snippet in Java

```java
static Tuple[] sortArray(Tuple[] uns){
    return Arrays.sort(
      uns, new Comparator<Tuple>() {
       public int compare(
        Tuple o1, Tuple o2) {
           return o1.get(0) == o2.get(0);
        }
    });
}
```

Summary: sort a list of tuples by first element

Figure 1: Example motivating the need to understand the association of program and natural languages for code summarization, generation, and translation.

Note that the use of NL in software development is quite different than colloquially written and spoken language. For example, NL in software development often contains domain-specific jargon, *e.g.,* when software engineers use *Code Smell*[1], it means a potential problem in code (something other than *Smell* in regular English language).

In this work, our goal is to develop a general-purpose model that can be used in various PLUG applications. Recent advancements in deep learning and the availability of large-scale PL and developers' NL data ushered in the automation of PLUG applications. One important aspect of PLUG applications is that they demand a profound understanding of program syntax and semantics and mutual dependencies between PL and NL. For example, Figure 1 shows two implementations of the same algorithm (sorting) in two PL and corresponding NL summary. An automatic translation tool must understand that function `sorted` in Python acts similar to `Arrays.sort` in Java and the `lambda`

*Equal contribution.

[1]https://en.wikipedia.org/wiki/Code_smell

Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2655–2668
June 6–11, 2021.

operation in Python is equivalent to instantiating a `Comparator` object in Java. Similarly, a tool that summarizes either of these code must understand that `x[0]` in Python or `Tuple.get(0)` in Java refers to the first element in the tuple list.

Most of the available data in PL and NL are unlabeled and cannot be trivially used to acquire PLUG task-specific supervision. However, PLUG tasks have a common prerequisite — understanding PL and NL syntax and semantics. Leveraging unlabelled data to pretrain a model to learn PL and NL representation can be transferred across PLUG tasks. This approach reduces the requirement of having large-scale annotations for task-specific fine-tuning. In recent years we have seen a colossal effort to pretrain models on a massive amount of unlabeled data (*e.g.,* text, images, videos) (Devlin et al., 2019; Liu et al., 2019; Conneau and Lample, 2019; Conneau et al., 2020; Li et al., 2019; Sun et al., 2019) to transfer representation encoders across a wide variety of applications. There are a few research effort in learning general purpose PL-NL representation encoders, such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021) that are pretrained on a *small-scale* bimodal data (code-text pairs). Such models have been found effective for PLUG tasks, including code search, code completion, etc.

Language generation tasks such as code summarization is modeled as sequence-to-sequence learning, where an encoder learns to encode the input code and a decoder generates the target summary. Despite the effectiveness of existing methods, they do not have a pretrained decoder for language generation. Therefore, they still require a large amount of parallel data to train the decoder. To overcome this limitation, Lewis et al. (2020) proposed denoising sequence-to-sequence pre-training where a Transformer (Vaswani et al., 2017) learns to reconstruct an original text that is corrupted using an arbitrary noise function. Very recently, Lachaux et al. (2020) studied denoising pre-training using a large-scale source code collection aiming at unsupervised program translation and found the approach useful. This raises a natural question, *can we unify pre-training for programming and natural language?* Presumably, to facilitate such pre-training, we need unlabeled NL text that is relevant to software development. Note that unlike other bimodal scenarios (*e.g.,* vision and language), PL and associated NL text share the same alphabet or uses anchor tokens (*e.g.,* "sort", "list", "tuple" as shown in Figure 1) that can help to learn alignment between semantic spaces across languages.

	Java	Python	NL
All Size	352 GB	224 GB	79 GB
All - Nb of tokens	36.4 B	28 B	6.7 B
All - Nb of documents	470 M	210 M	47 M

Table 1: Statistics of the data used to pre-train PLBART. "Nb of documents" refers to the number of functions in Java and Python collected from Github and the number of posts (questions and answers) in the natural language (English) from StackOverflow.

We introduce PLBART (Program and Language BART), a bidirectional and autoregressive transformer pre-trained on unlabeled data across PL and NL to learn multilingual representations applicable to a broad spectrum of PLUG applications. We evaluate PLBART on code summarization, generation, translation, program repair, clone detection, and vulnerability detection tasks. Experiment results show that PLBART outperforms or rivals state-of-the-art methods, *e.g.,* CodeBERT and GraphCodeBERT, demonstrating its promise on program understanding and generation. We perform a thorough analysis to demonstrate that PLBART learns program syntax, logical data flow that is indispensable to program semantics, and excels even when limited annotations are available. We release our code[2] to foster future research.

2 PLBART

PLBART uses denoising sequence-to-sequence pre-training to utilize unlabeled data in PL and NL. Such pre-training lets PLBART reason about language syntax and semantics. At the same time, PLBART learns to generate language coherently.

2.1 Denoising Pre-training

Data & pre-processing We pre-train PLBART on a large-collection of Java and Python functions and natural language descriptions from Github and StackOverflow, respectively. We download all the GitHub repositories associated with Java and Python languages available on Google BigQuery.[3] We extract the Java and Python functions following the pre-processing pipeline from Lachaux et al. (2020). We collect the StackOverflow posts (include both questions and answers, exclude code

[2] https://github.com/wasiahmad/PLBART

[3] https://console.cloud.google.com/marketplace/details/github/github-repos

PLBART Encoder Input	PLBART Decoder Output
Is 0 the **[MASK]** Fibonacci **[MASK]** ? <En>	<En> Is 0 the **first** Fibonacci **number** ?
public static main (String args [ ]) { date = Date ( ) ; System . out . (String . format (" Current Date : % tc " ,)) ; } <java>	<java> public static **void** main (String args [ ]) { **Date** date = **new** Date ( ) ; System . out . **printf** (String . format (" Current Date : % tc " , **date**)) ; }
def addThreeNumbers (x , y , z) : NEW_LINE INDENT return **[MASK]** <python>	<python> def addThreeNumbers (x , y , z) : NEW_LINE INDENT return **x + y + z**

Table 2: Example encoder inputs and decoder outputs during denoising pre-training of PLBART. We use three noising strategies: token masking, token deletion, and token infilling (shown in the three examples, respectively).

snippets) by downloading the data dump (date: 7th September 2020) from stackexchange.[4] Statistics of the pre-training dataset are presented in Table 1. We tokenize all the data with a sentencepiece model (Kudo and Richardson, 2018) learned on 1/5'th of the pre-training data. We train sentencepiece to learn 50,000 subword tokens.

One key challenge to aggregate data from different modalities is that some modalities may have more data, such as we have 14 times more data in PL than NL. Therefore, we mix and up/down sample the data following Conneau and Lample (2019) to alleviate the bias towards PL. We sample instances for pre-training according to a multinomial distribution with probabilities $(q_1, q_2, \ldots, q_N)$:

$$q_i = \frac{1}{p_i} \cdot \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha}, p_i = \frac{n_i}{\sum_{j=1}^{N} n_j},$$

where N is the total number of languages and n_i is the total number of instances in language i. We set the smoothing parameter α to 0.3.

Architecture PLBART uses the same architecture as BART$_{base}$ (Lewis et al., 2020), it uses the sequence-to-sequence Transformer architecture (Vaswani et al., 2017), with 6 layers of encoder and 6 layers of decoder with model dimension of 768 and 12 heads (~140M parameters). The only exception is, we include an additional layer-normalization layer on top of both the encoder and decoder following Liu et al. (2020), which is found to stabilize training with FP16 precision.

Noise function, f In denoising autoencoding, a model learns to reconstruct an input text that is corrupted by a noise function. Reconstruction of the original input requires the model to learn language syntax and semantics. In this work, we use three noising strategies: token masking, token deletion, and token infilling (Lewis et al., 2020). According to the first two strategies, random tokens are sampled and replaced with a mask token or deleted from the input sequence. In token infilling, a number of text spans are sampled and replaced with a *single* mask token. The span lengths are drawn from a Poisson distribution (λ = 3.5). We mask 35% of the tokens in each instance.

Input/Output Format The input to the encoder is a noisy text sequence, while the input to the decoder is the original text with one position offset. A language id symbol (e.g., <java>, <python>) is appended and prepended to the encoder and decoder inputs, respectively. We provide a few examples in Table 2. The input instances are truncated if they exceed a maximum sequence length of 512.

Learning PLBART is pre-trained on N languages (in our case, N=3), where each language N_i has a collection of unlabeled instances $\mathcal{D}_i = \{x_1, \ldots, x_{n_i}\}$. Each instance is corrupted using the noise function f and we train PLBART to predict the original instance x from $f(x)$. Formally, PLBART is trained to maximize $\mathcal{L}_\theta$:

$$\mathcal{L}_\theta = \sum_{i=1}^{N} \sum_{j=1}^{m_i} \log P(x_j | f(x_j); \theta)$$

where m_i is the number of sampled instances in language i and the likelihood P is estimated following the standard sequence-to-sequence decoding.

Optimization We train PLBART on 8 Nvidia GeForce RTX 2080 Ti GPUs for 100K steps. The effective batch size is maintained at 2048 instances. We use Adam (ϵ = 1e-6, β_2 = 0.98) with a linear learning rate decay schedule for optimization. We started the training with dropout 0.1 and reduced it to 0.05 at 50K steps and 0 at 80K steps. This is done to help the model better fit the data (Liu et al., 2020). The total training time was approximately

[4] https://archive.org/download/stackexchange

	PLBART Encoder Input	PLBART Decoder Input
S	def maximum (a , b , c) : NEW_LINE INDENT return max ([a , b , c]) <python>	<En> Find the maximum of three numbers
G	Find the maximum of three numbers <En>	<java> public int maximum (int a , int b , int c) { return Math . max (a , Math . max (b , c)) }
T	public int maximum (int a , int b , int c) { return Math . max (a , Math . max (b , c)) } <java>	<python> def maximum (a , b , c) : NEW_LINE INDENT return max ([a , b , c])

Table 3: Example inputs to the encoder and decoder for fine-tuning PLBART on sequence generation tasks: source code summarization (S), generation (G), and translation (T).

276 hours (11.5 days). All experiments are done using the Fairseq library (Ott et al., 2019).

2.2 Fine-tuning PLBART

We fine-tune PLBART for two broad categories of downstream applications.

Sequence Generation PLBART has an encoder-decoder architecture where the decoder is capable of generating target sequences autoregressively. Therefore, we can directly fine-tune PLBART on sequence generation tasks, such as code summarization, generation, and translation. Unlike denoising pre-training, the source sequence is given as input to the encoder during fine-tuning, and the decoder generates the target sequence. The source and target sequence can be a piece of code or text sequence. Table 3 shows a few examples of input and output to and for PLBART for different generation tasks. Note that PLBART prepends a language id to the decoded sequence; it enables fine-tuning PLBART in a multilingual setting (*e.g.,* code generation in multiple languages).[5]

Sequence Classification We fine-tune PLBART on sequence classification tasks following Lewis et al. (2020). The input sequence is fed into both the encoder and decoder. For a pair of inputs, we concatenate them but insert a special token ("</s>") between them. A special token is added at the end of the input sequence. This last token's representation from the final decoder layer is fed into a linear classifier for prediction.

Optimization We fine-tune PLBART for a maximum of 100K steps on all the downstream tasks with 2500 warm-up steps. We set the maximum learning rate, effective batch size, and dropout rate to 3e-5, 32 and 0.1, respectively. The final models are selected based on the validation BLEU (in generation task) or accuracy (in classification tasks). Fine-tuning PLBART is carried out in one Nvidia GeForce RTX 2080 Ti GPU.

[5]We do not perform multilingual fine-tuning in this work.

3 Experiment Setup

To understand PLBART's performance in a broader context, we evaluate PLBART on several tasks. Our evaluation focuses on assessing PLBART's ability to capture rich semantics in source code and associated natural language text.

3.1 Evaluation Tasks

We divide the evaluation tasks into four categories. The evaluation task datasets are summarized in Table 4. We use CodeXGLUE (Lu et al., 2021) provided public dataset and corresponding train-validation-test splits for all the tasks.

Code Summarization refers to the task of generating a natural language (English) summary from a piece of code. We fine-tune PLBART on summarizing source code written in six different programming languages, namely, Ruby, Javascript, Go, Python, Java, and PHP.

Code Generation is exactly the opposite of code summarization. It refers to the task of generating a code (in a target PL) from its NL description. We fine-tune PLBART on the Concode dataset (Iyer et al., 2018), where the input is a text describing class member functions in Java and class environment, the output is the target function.

Code Translation requires a model to generate an equivalent code in the target PL from the input code written in the source PL. Note that the source and target PL can be the same. Hence, we consider two types of tasks in this category.

The first task is a typical PL translation task, translating a code *i.e.,* from Java code to C#, and vice versa. In this task, the semantic meaning of the translated code should exactly match the input

Task	Dataset	Language	Train	Valid	Test
Summarizaion	Husain et al. (2019)	Ruby	24,927	1,400	1,261
		Javascript	58,025	3,885	3,291
		Go	167,288	7,325	8,122
		Python	251,820	13,914	14,918
		Java	164,923	5,183	10,955
		PHP	241,241	12,982	14,014
Generation	Iyer et al. (2018)	NL to Java	100,000	2,000	2,000
Translation	Code-Code (Lu et al., 2021)	Java to C#	10,300	500	1,000
		C# to Java	10,300	500	1,000
	Program Repair (Tufano et al., 2019)	Java_{small}	46,680	5,835	5,835
		Java_{medium}	52,364	6,545	6,545
Classification	Vulnerability Detection (Zhou et al., 2019)	C/C++	21,854	2,732	2,732
	Clone Detection (Wang et al., 2020)	Java	100,000	10,000	415,416

Table 4: Statistics of the downstream benchmark datasets.

code. Thus, this task evaluates PLBART's understanding of program semantics and syntax across PL. The second task we consider is program repair. In this task, the input is a buggy code, and the output is a modified version of the same code which fixes the bug. This task helps us understand PLBART's ability to understand code semantics and apply semantic changes in the code.

Code Classification aims at predicting the target label given a single or a pair of source code. We evaluate PLBART on two classification tasks. The first task is clone detection, where given a pair of code, the goal is to determine whether they are clone of each other (similar to paraphrasing in NLP). The second task is detecting whether a piece of code is vulnerable. This task help us gauging PLBART's effectiveness in program understanding in an unseen PL since the code examples in this task are written in C/C++.

3.2 Evaluation Metrics

BLEU computes the n-gram overlap between a generated sequence and a collection of references. We use corpus level BLEU (Papineni et al., 2002) score for all the generation tasks, except code summarization where we use smoothed BLEU-4 score (Lin and Och, 2004) following Feng et al. (2020).

CodeBLEU is a metric for measuring the quality of the synthesized code (Ren et al., 2020). Unlike BLEU, CodeBLEU also considers grammatical and logical correctness based on the abstract syntax tree and the data-flow structure.

Exact Match (EM) evaluates if a generated sequence exactly matches the reference.

3.3 Baseline Methods

We compare PLBART with several state-of-the-art models and broadly divide them into two categories. First, the models that are trained on the evaluation tasks from scratch, and second, the models that are pre-trained on unlabeled corpora and then fine-tuned on the evaluation tasks.

3.3.1 Training from Scratch

Seq2Seq (Luong et al., 2015) is an LSTM based Seq2Seq model with attention mechanism. Vocabulary is constructed using byte-pair encoding.

Transformer (Vaswani et al., 2017) is the base architecture of PLBART and other pre-trained models. Transformer baseline has the same number of parameters as PLBART. Hence, a comparison with this baseline demonstrates the direct usefulness of pre-training PLBART.

3.3.2 Pre-trained Models

As described in section 2, PLBART consists of an encoder and autoregressive decoder. We compare PLBART on two categories of pre-trained models. First, the encoder-only models (*e.g.,* RoBERTa, CodeBERT, and GraphCodeBERT) that are combined with a randomly initialized decoder for task-specific fine-tuning. The second category of baselines include decoder-only models (CodeGPT) that can perform generation autoregressively.

Methods	Ruby	Javascript	Go	Python	Java	PHP	Overall
Seq2Seq	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer	11.18	11.59	16.38	15.81	16.26	22.12	15.56
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	**25.16**	17.83
PLBART	**14.11**	**15.56**	**18.91**	**19.30**	**18.45**	23.58	**18.32**

Table 5: Results on source code summarization, evaluated with smoothed BLEU-4 score. The baseline results are reported from Feng et al. (2020).

RoBERTa, RoBERTa (code) are RoBERTa (Liu et al., 2019) model variants. While RoBERTa is pre-trained on natural language, RoBERTa (code) is pre-trained on source code from CodeSearchNet (Husain et al., 2019).

CodeBERT (Feng et al., 2020) combines masked language modeling (MLM) (Devlin et al., 2019) with replaced token detection objective (Clark et al., 2020) to pretrain a Transformer encoder.

GraphCodeBERT (Guo et al., 2021) is a concurrent work with this research which improved CodeBERT by modeling the data flow edges between code tokens. We report GraphCodeBERT's performance directly from the paper since their implementation is not publicly available yet.

GPT-2, CodeGPT-2, and CodeGPT-adapted are GPT-style models. While GPT-2 (Radford et al., 2019) is pretrained on NL corpora, CodeGPT-2 and CodeGPT-adapted are pretrained on CodeSearchNet (Lu et al., 2021). Note that, CodeGPT-adapted starts from the GPT-2 checkpoint for pre-training.

4 Results & Analysis

We aim to address the following questions.

1. Does PLBART learn strong program and language representations from unlabeled data?
2. Does PLBART learn program characteristics, *e.g.,* syntax, style, and logical data flow?
3. How does PLBART perform in an unseen language with limited annotations?

4.1 Code Summarization

Table 5 shows the result of code summarization. PLBART outperforms the baseline methods in five out of the six programming languages with an overall average improvement of 0.49 BLEU-4 over CodeBERT. The highest improvement (~16%) is in the Ruby language, which has the smallest amount of training examples. Unlike CodeBERT, PLBART is not pretrained on the Ruby language; however, the significant performance improvement indicates that PLBART learns better generic program semantics. In contrast, PLBART performs poorly in the PHP language. The potential reason is syntax mismatch between the pre-trained languages and PHP. Surprisingly, RoBERTa performs better than PLBART on the PHP language. We suspect that since RoBERTa is pre-trained on natural language only, it does not suffer from the syntax mismatch issue. Overall in comparison to the Transformer baseline, PLBART improves with an average of 2.76 BLEU-4, and we credit this improvement to the pre-training step.

Methods	EM	BLEU	CodeBLEU
Seq2Seq	3.05	21.31	26.39
Guo et al. (2019)	10.05	24.40	29.46
Iyer et al. (2019)	12.20	26.60	-
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	**20.10**	32.79	35.98
PLBART	18.75	**36.69**	**38.52**
PLBART_{10K}	17.25	31.40	33.32
PLBART_{20K}	18.45	34.00	35.75
PLBART_{50K}	17.70	35.02	37.11

Table 6: Results on text-to-code generation task using the CONCODE dataset (Iyer et al., 2018).

4.2 Code Generation

Table 6 shows the evaluation result on code generation from NL description. PLBART outperforms all the baselines in terms of BLEU and CodeBLEU. While CodeGPT-adapted (Lu et al., 2021) achieves the best Exact Match (EM) score, PLBART outperforms CodeGPT-adapted by a large margin in terms of CodeBLEU. This result implies that PLBART generates *significantly more* syntactically and logically correct code than all the baselines.

Figure 2 shows an example of code generated by PLBART. The difference between the reference code and the generated code is in line 6 onward. In the reference code, `loc0` is returned, however

Methods	Java to C#			C# to Java		
	BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU
Naive Copy	18.54	0	34.20	18.69	0	43.04
PBSMT	43.53	12.50	42.71	40.06	16.10	43.48
Transformer	55.84	33.00	63.74	50.47	37.90	61.59
RoBERTa (code)	77.46	56.10	83.07	71.99	57.90	80.18
CodeBERT	79.92	59.00	85.10	72.14	58.80	79.41
GraphCodeBERT	80.58	59.40	-	72.64	58.80	-
PLBART	**83.02**	**64.60**	**87.92**	**78.35**	**65.00**	**85.27**

Table 7: Results on source code translation using Java and C# language dataset introduced in (Lu et al., 2021). PBSMT refers to phrase-based statistical machine translation where the default settings of Moses decoder (Koehn et al., 2007) is used. The training data is tokenized using the RoBERTa (Liu et al., 2019) tokenizer.

Input text: returns the count to which the specified key is mapped in this frequency counter , or 0 if the map contains no mapping for this key .

Reference Code

```
1 Integer function (T arg0) {
2    Integer loc0 = counter.get(arg0);
3    if (loc0 == null) {
4       return 0 ;
5    }
6    return loc0;
7 }
```

Generated Code

```
1 int function (T arg0) {
2    Integer loc0 = counter.get(arg0);
3    if (loc0 == null) {
4       return 0 ;
5    }
6    else {
7       return loc0;
8    }
9 }
```

Figure 2: An example of generated code by PLBART that is syntactically and semantically valid, but does not match the reference.

same `loc0` is returned in an `else` block in the generated code. If we look closely, in the reference code, line 6 will be executed only if the condition in line 3 (*i.e.,* `loc0 == null`) is `false`. In the generated code, `loc0` will be returned only if the condition in line 3 is `false`, making the generated code semantically equivalent to the reference code.

To study whether PLBART learns code syntax and logical flow during pre-training or fine-tuning, we perform an ablation study where we use subset of the training examples (10K, 20K, and 50K) to fintune PLBART in this task. As table 6 shows, with only 10K examples, PLBART outperforms all baselines in terms of CodeBLUE. This ablation shows that PLBART learns program syntax and data flow during pre-training, resulting in effective performance on downstream tasks even when finetuned on small number of examples.

As shown in prior works (Yin and Neubig, 2017; Chakraborty et al., 2020), generating syntactically and logically correct code has been a big challenge in program generation. We conjecture that PLBART's large-scale denoising sequence-to-sequence pre-training helps understand program syntax and logical flow; therefore enables PLBART to generate syntactically and logically valid code.

4.3 Code Translation

Table 7 presents the evaluation results on code translation. PLBART outperforms all the baselines *w.r.t.* EM, BLEU, and CodeBLEU. PLBART improves over CodeBERT by 9.5% and 10.5% when translating from Java to C# and C# to Java, respectively. Although PLBART is not pretrained on C# language, there is a significant syntactic and semantic similarity between Java and C#. Thus PLBART understands C# language syntax and semantics. However, such similarities are non-trivial, making the Naive copy and PBSMT perform very poorly in both the translation tasks.

Figure 3 shows an example where PLBART's generated C# code does not exactly match the reference; however, they are semantically equivalent. In the reference, the `else` block (line 4-9) is equivalent to the `else if` block (line 4-7) in the generated code. In addition, `start` is generated as function parameter and used in the function body, equivalent to `start_1` in the reference code. This further corroborates the syntactic understanding of PLBART and its ability to reason about the data flow in source code. We present more qualitative examples in Appendix.

Reference Code : C#

```
public bool find(int start_1){
    findPos = start_1;
    ...
    else{
        if (findPos >= _regionEnd){
            matchFound = false;
            return false;
        }
    }
    ...
}
```

Generated Code : C#

```
public bool find(int start){
    findPos = start;
    ...
    else if (findPos >= _regionEnd){
        matchFound = false;
        return false;
    }
    ...
}
```

Figure 3: Example C# code generated by PLBART that does not exactly match the reference code.

In the program repair task, both the input and the output are in the same language. While the input is a buggy code, the output should be the target bug-free code. Thus in this task, the exact match is the critical metric. Nevertheless, as shown in table 8, PLBART can generate 17.13%, and 74.03% more correct bug fixes than CodeBERT in Java$_{small}$ and Java$_{medium}$ datasets, respectively. On the other hand, PLBART performs comparably to GraphCodeBERT that uses structure-aware pre-training to learn program syntax and semantics.

4.4 Classification

In both clone detection and the vulnerability detection tasks, PLBART outperforms CodeBERT. We present the results in Table 9. In the vulnerability detection task, code semantics is the most critical feature (Zhou et al., 2019; Chakraborty et al., 2020). Since PLBART is not pretrained on C/C++ language, its improved performance compared to the Transformer baseline is the testament that PLBART can identify semantics beyond the language syntax's specifics. Moreover, PLBART's improved performances over CodeBERT and GraphCodeBERT confirms its effectiveness in program understanding in addition to its generation ability.

We acknowledge that neither PLBART nor CodeBERT is state-of-the-art in vulnerability detection, as graph-based models perform best in this task.

Methods	Small		Medium	
	EM	BLEU	EM	BLEU
Naive Copy	0	78.06	0	90.91
Seq2Seq	10.00	76.76	2.50	72.08
Transformer	14.70	77.21	3.70	89.25
CodeBERT	16.40	77.42	5.16	**91.07**
GraphCodeBERT	17.30	**80.58**	**9.10**	72.64
PLBART	**19.21**	77.02	8.98	88.50

Table 8: Results on program repair (in Java).

Tasks	Vulnerability Detection	Clone Detection
Transformer	61.64	-
CodeBERT	62.08	96.5
GraphCodeBERT	-	97.1
PLBART	**63.18**	**97.2**

Table 9: Results on the vulnerable code detection (accuracy) and clone detection (F1 score) tasks.

In this evaluation, our goal is to study how well PLBART understands program semantics in an unseen language for a different type of task (other than the generation, *i.e.,* classification).

5 Related Work

Pre-training for Language Understanding and Generation Transformer (Vaswani et al., 2017), a sequence-to-sequence architecture that includes an encoder and decoder, has shown tremendous promise in natural language processing (NLP), computer vision, software engineering, and more. Devlin et al. (2019) first proposed to pre-train a large Transformer architecture, called BERT, to learn representations of natural language using large-scale unlabeled data in a self-supervised fashion. Later, BERT's task-independent pre-training approach is rigorously studied (Devlin et al., 2019; Liu et al., 2019; Solaiman et al., 2019; Feng et al., 2020; Sun et al., 2019; Li et al., 2020). While BERT-like models have shown effectiveness in learning contextualized representation, it is not very useful in generation tasks. GPT (Radford et al., 2018) style models improve upon BERT for generative tasks with autoregressive pre-training; however, unlike BERT, they are not bidirectional. Lewis et al. (2020) introduced BART, a denoising autoencoder that uses a bidirectional encoder and an auto-regressing decoder. Similar to BART, PLBART uses denoising pre-training to cope with generative tasks and learns multilingual representations of programming and natural language jointly.

Deep Learning in Software Engineering There is a growing interest in automating software engineering (SE) using deep learning in the last few years. Vast sources of code in open source repositories and forums make deep learning feasible for SE tasks. Code Summarization (Movshovitz-Attias and Cohen, 2013; Allamanis et al., 2016; Iyer et al., 2016; Alon et al., 2019a; Hu et al., 2018; Harer et al., 2019; Ahmad et al., 2020), Bug Detection (Ray et al., 2016; Li et al., 2018b; Russell et al., 2018; Zhou et al., 2019; Chakraborty et al., 2020), Program Repair (Chen et al., 2019; Chakraborty et al., 2020; Lutellier et al., 2020), Code Translation (Chen et al., 2018; Drissi et al., 2018; Xu et al., 2020), Clone Detection (Zhang et al., 2019; Yu et al., 2019; Wang et al., 2020), Code completion (Li et al., 2018a; Hellendoorn and Devanbu, 2017; Parvez et al., 2018) are some of the tasks that are addressed with deep neural solution. While most of the prior approaches use task-specific representation learning, a few works (Alon et al., 2019b; Feng et al., 2020; Guo et al., 2021; Lachaux et al., 2020; Clement et al., 2020) attempted to learn transferable representations in an unsupervised fashion. More closely to our work, CodeBERT (Feng et al., 2020) is pre-trained on bimodal data to capture the semantic interaction between the input modalities (*i.e.,* program and natural languages). More recently, GraphCodeBERT (Guo et al., 2021) improves upon CodeBERT by leveraging data flow in source code. In contrast, PLBART is pre-trained on large-scale data using denoising autoencoding to learn the program and natural language representations that make it effective for a broad spectrum of software engineering tasks.

6 Conclusion

This paper presents PLBART, a sizeable pre-trained sequence-to-sequence model that can perform program and language understanding and generation tasks. PLBART achieves state-of-the-art performance on various downstream software engineering tasks, including code summarization, code generation, and code translation. Furthermore, experiments on discriminative tasks establish PLBART's effectiveness on program understanding. We also show that PLBART learns crucial program characteristics due to pre-training, such as syntax, identifier naming conventions, data flow. In the future, we want to explore ways to fine-tune PLBART on all the downstream tasks jointly.

Broader Impact

Automation in software engineering is paramount in increasing programmers' productivity. A reduced workload of tedious works at the part of developers' daily routine would give them more time to solve significant problems for society's wellbeing. There are numerous program-and-language applications in the software development lifecycle, such as code documentation/summarization, code synthesis, translating code across languages, etc that can be automated to facilitate software engineering. The availability of large-scale data (thanks to open source repositories, forums, and millions of contributors worldwide) opens up the opportunity to solve many of those problems in a data-driven fashion. PLBART aims at program-and-language applications that demand a complete syntactic and semantic understanding of source code and associated textual data. For the tasks we have shown evaluation, PLBART will serve as a solid and replicable baseline to guide future research. We also believe our work could be an excellent starting point for future works aim at solving a variety of software engineering problems.

Acknowledgments

We thank anonymous reviewers for their helpful feedback. We also thank UCLA-NLP group for helpful discussions and comments. This work was supported in part by National Science Foundation Grant OAC 1920462, CCF 1845893, CCF 1822965, CNS 1842456. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or NSF.

References

Wasi 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*, pages 4998–5007, Online. Association for Computational Linguistics.

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, page 40. ACM.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1–1.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep learning based vulnerability detection: Are we there yet? *arXiv preprint arXiv:2009.07235*.

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.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online. Association for Computational Linguistics.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.

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.

Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. 2018. Program language translation using a grammar-driven tree-to-tree model. In *ICML Workshop on Neural Abstract Machines & Program Induction (NAMPI v2)*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. 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.

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*.

Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2019. Coupling retrieval and meta-learning for context-dependent semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 855–866, Florence, Italy. Association for Computational Linguistics.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, page 200–210, New York, NY, USA. Association for Computing Machinery.

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*.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. 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 5426–5435, Hong Kong, China. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

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.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018a. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4159–4165. International Joint Conferences on Artificial Intelligence Organization.

Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. 2019. Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557*.

Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. 2020. What does BERT with vision look at? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5265–5275, Online. Association for Computational Linguistics.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018b. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

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*.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–114, New York, NY, USA. Association for Computing Machinery.

Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational*

Linguistics (Volume 2: Short Papers), pages 35–40, Sofia, Bulgaria. Association for Computational Linguistics.

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, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439, New York, NY, USA. Association for Computing Machinery.

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*.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*.

Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. 2019. Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7464–7473.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

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.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE.

Haoran Xu, Shuhui Fan, Yongjun Wang, Zhijian Huang, Hongzuo Xu, and Peidai Xie. 2020. Tree2tree structural language modeling for compiler fuzzing. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 563–578. Springer International Publishing.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80. IEEE Press.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 783–794. IEEE Press.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32, pages 10197–10207. Curran Associates, Inc.

Hyper-parameter	RoBERTa*	CodeGPT-2	CodeBERT	GraphCodeBERT	PLBART
vocab size	50,265	50,001	50,265	-	50,004
n_positions	514	1024	514	256	1024
model size	768	768	768	768	768
# layers	12	12	12	12	6
# heads	12	12	12	12	12
d_{ff}	3072	3072	3072	-	3072
dropout	0.1	0.1	0.1	-	0.1
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	5e-5	5e-5	5e-5	1e-4	5e-5
batch size	32	32	32	64	32

Table 10: Details of the hyper-parameters used during fine-tuning for sequence generation tasks. * indicates pre-trained from scratch using source code-text pairs.

Example 1 : get the msg value.

Reference Code

```
String function() {
    return msg;
}
```

Generated Code

```
String function() {
    return this.msg;
}
```

Example 2 : returns the instance of the singleton .

Reference Code

```
IConfigurationFactory function() {
  return SINGLETON;
}
```

Generated Code

```
IConfigurationFactory function() {
  if (SINGLETON == null){
    SINGLETON = new SINGLETON();
  }
  return SINGLETON;
}
```

Example 3 : convert the reader into an inputstream .

Reference Code

```
InputStream function () {
  return new ReaderInputStream(reader);
}
```

Generated Code

```
InputStream function () {
  return reader.getInputStream();
}
```

Example 4 : setter for a property . if the property already exists , the value will be overridden .

Reference Code

```
void function (
    final String arg0,
    final String arg1 ) {
    properties.setProperty (
       arg0, arg1) ;
}
```

Generated Code

```
void function (
    String arg0,
    String arg1) {
    properties.put (
       arg0, arg1) ;
}
```

Example 5 : clear the buffer .

Reference Code

```
void function() {
    bufferSize = 0;
}
```

Generated Code

```
void function() {
    buffer = null;
}
```

Figure 4: Qualitative examples of "text to code" generation using PLBART, where generated code does not exactly match the reference.

Example 1 : ✔

Input Code : C#

```
public int GetCells(){
    int size = 0;
    foreach (char c in cells.Keys){
        Cell e = At(c);
        if (e.cmd >= 0 || e.@ref >= 0){
            size++;
        }
    }
    return size;
}
```

Generated Code : Java

```
public int getCells() {
    Iterator<Character> i =
      cells.keySet().iterator();
    int size = 0;
    for (; i.hasNext();) {
        Character c = i.next();
        Cell e = at(c);
        if (e.cmd >= 0 || e.ref >= 0){
            size++;
        }
    }
    return size;
}
```

Example 2 : ✔

Input Code : C#

```
public override Cell Merge(
  Cell m, Cell e){
   if (m.cmd == e.cmd && m.@ref ==
    e.@ref && m.skip == e.skip) {
       Cell c = new Cell(m);
       c.cnt += e.cnt;
       return c;
    }
    else {
       return null;
    }
}
```

Generated Code : Java

```
public Cell merge(Cell m, Cell e) {
    if (m.cmd == e.cmd &&
        m.ref == e.ref &&
        m.skip == e.skip) {
        Cell c = new Cell(m);
        c.cnt += e.cnt;
        return c;
    }
    else {
        return null;
    }
}
```

Example 3 : ✔

Input Code : Java

```
@Override public List<E> subList(
  int start, int end) {
   synchronized (mutex) {
      return new
        SynchronizedRandomAccessList
        <E>(
         list.subList(start, end),
         mutex);
    }
}
```

Generated Code : C#

```
public override java.util.List<E>
  SubList(int start, int end){
   lock (mutex){
      return new java.util.Collections.
        SynchronizedRandomAccessList<E>(
         list.subList(start, end),
         mutex
      );
    }
}
```

Example 4 : ✔

Input Code : Java

```
public
  PersianNormalizationFilterFactory(
      Map<String,String> args) {
   super(args);
   if (!args.isEmpty()) {
      throw
        new IllegalArgumentException(
          "Unknown parameters: "
          + args);
    }
}
```

Generated Code : C#

```
public
  PersianNormalizationFilterFactory(
   IDictionary<string, string> args)
       : base(args){
   if (args.Count > 0){
      throw new System.ArgumentException(
         "Unknown parameters: "
         + args
      );
    }
}
```

Figure 5: Qualitative examples of source code translation between Java and C# using PLBART.