Regular Paper

NMT-Based Code Generation for Coding Assistance with Natural Language

Yuka Akinobu[1,†1,a] Teruno Kajiura[1] Momoka Obara[1] Kimio Kuramitsu[2,b]

Received: August 31, 2021, Accepted: January 11, 2022

Abstract: This paper proposes an attempt to realize coding assistance that generates Python code from natural language descriptions using neural machine translation. Although coding assistance with deep learning has recently become a major concern, few applications have used neural machine translation models. One of the major barriers is the shortage of a parallel corpus of natural language descriptions and source code. To overcome the shortage of parallel corpora, we propose a method for synthesizing parallel corpora that utilizes the formal nature of programming languages. We aim to establish a new method using an abstract syntax tree (AST) and a corpus of code fragments. Using the proposed synthesis method, we successfully constructed tens of thousands of parallel corpora and trained PyNMT models to generate Python code from Japanese input sentences. The trained PyNMT models successfully predicted the Python code from user input sentences with an accuracy of 28%. In this study, we propose a synthetic method for a parallel corpus and summarize the results of the evaluation experiments conducted on PyNMT models.

Keywords: neural machine translation, code generation, back-translation

1. Introduction

Programming is changing from a specialized skill used in software development to an essential skill such as data analysis and visualization. However, programming is not so simple that anyone can easily master and use because it requires learning the language syntax and many of APIs which the programmer need to use. Particularly, it is hard for many Japanese speakers who have difficulty with English to remember English-based API names, which can lead to a major barrier to programming.

To reduce such barriers, we aimed at realizing coding assistance using a natural language. One of the features of our approach is the use of a neural machine translation (NMT) to generate code from natural language descriptions. We believe the user can intuitively understand the NMT-generated Python code, because the code is relatively short.

More recently, coding assistance based on deep learning has attracted a significant interest especially among software development companies, and various research attempts have been made and the resulting products have appeared on the market. However, coding assistance with NMT is rare. The main reason for this is the absolute lack of a parallel code corpus that serves as the training data. In machine translation between natural languages such as Japanese–English translation, tens of millions of parallel corpora have been developed as data resources. However, the parallel code corpus between a natural language and the source code is limited; approximately 14,000 in Django (English–Python), less than 3,000 in CoNaLa (English–Python), and approximately 700 in the Euler corpus (Japanese–Python). With such a small parallel code corpus, it is difficult to build an NMT model with sufficient accuracy, even when using the most advanced deep learning techniques.

In this paper, we propose novel back-translation (BT) methods for synthesizing parallel code corpora that utilize the formal properties of programming languages to solve the shortage of parallel code corpora. BT is a technique for generating bilingual translations by building a translation model that reverses the input and output of the designated model. The technique is used as a data augmentation method in machine translation. In this study, two methods are proposed, i.e., a top-down BT method, which synthesizes Japanese from an AST of the source code, and a bottom-up BT method, which synthesizes multiple corpora from the corpus of code fragments in expression units. Using these synthetic methods, we constructed tens of thousands of parallel code corpora and built the PyNMT model to generate appropriate Python code from Japanese descriptions.

This paper shows the proposed methods for synthesizing parallel code corpora and report their experiment results. The remainder of this paper is organized as follows. Section 2 describes the motivation of our study. Section 3 provides an overview of the NMT and parallel code corpus. Section 4 describes the top-down and bottom-up BT approaches. Section 5 describes the experiment evaluation of the PyNMT model trained with parallel code corpora constructed using the proposed methods. Section 6 reviews related works, and Section 7 provides some concluding remarks.

[1] Division of Mathematical and Physical Sciences, Graduate School of Science, Japan Women's University, Bunkyo, Tokyo 112–8681, Japan
[2] Department of Mathematics, Physics, and Computer Science, Japan Women's University, Bunkyo, Tokyo 112–8681, Japan
[†1] Presently with NTT Software Innovation Center
This work has been done when the first author was at the Japan Women's University
[a] m1616003ay@ug.jwu.ac.jp
[b] kuramitsuk@fc.jwu.ac.jp

2. Motivation

The concept behind this study is derived from a situation where many programming beginners are hard to remember API usages to implement for a particular process. Specifically in data analysis, which requires the use of numerous libraries, such as Pandas and Matplotlib, search engines are often used to find references and sample codes. However, such task is tedious and cumbersome, because it requires considerable time and experience.

To solve this problem, we have envisioned a novel coding assistance system that generates an appropriate source code based on the intentions described in natural language. **Figure 1** illustrates the concept of the envisioned coding assistance system. The key idea is the code translation from natural language into hard-to-remember code. We expect to provide users with a seamless coding experience by incorporating natural language into their coding assistance.

3. Neural Machine Translation and Code Generation

The PyNMT model, we have built, is an NMT model that outputs Python code from Japanese descriptions. NMT has significantly improved the translation accuracy since the advancement of deep learning technology in the 2010s, particularly with the advent of the deep learning model, Transformer [1]. In this section, we provide an overview of NMT and describe the parallel code corpus, which is an essential factor for determining the accuracy of the PyNMT model.

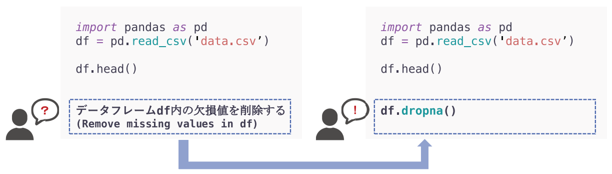
3.1 PyNMT

In recent years, NMT, or neural machine translation, is used as a key mechanism of machine translation apps and has achieved significant improvements in terms of accuracy in recent years [2]. **Figure 2** illustrates the training procedure of the PyNMT model M. To train the PyNMT model, we need to use an encoder–decoder model with a parallel code corpus, which pairs the source sentences X and target sentences Y. Using an encoder–decoder model, the PyNMT model can learn features between different languages. When the training of the PyNMT model is completed, it can predict the translated sentence Y' from the unknown source sentence X' that requires to be translated.

3.2 Transformer

Transformer is an encoder–decoder model proposed in 2017. It has succeeded in reflecting the dependencies between words more accurately and rapidly by effectively using different types of attention mechanisms. As a result, Transformer has achieved numerous SoTAs in natural language processing tasks such as machine translation and has contributed to the creation of many new language models [3], [4].

Figure 3 shows the architecture of Transformer. Transformer applies word embedding using a positional encoding layer in addition to the standard encoding layer to handle the order of the input strings without the use of CNNs or RNNs. Using the vector representation obtained, Transformer probabilistically predicts the next token that is most likely to occur.



```
import pandas as pd
df = pd.read_csv('data.csv’)

df.head()

データフレームdf内の欠損値を削除する
(Remove missing values in df)
```

```
import pandas as pd
df = pd.read_csv('data.csv’)

df.head()

df.dropna()
```

Fig. 1 Proposed concept of coding assistance system.

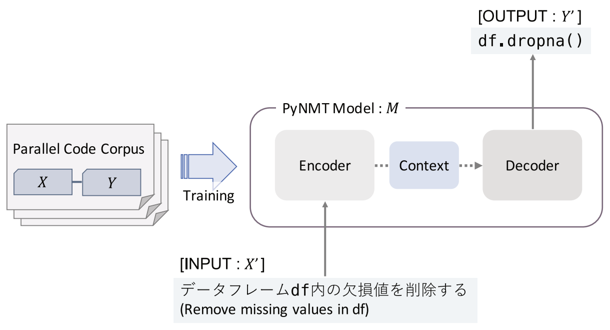


Fig. 2 Training procedure of PyNMT model.

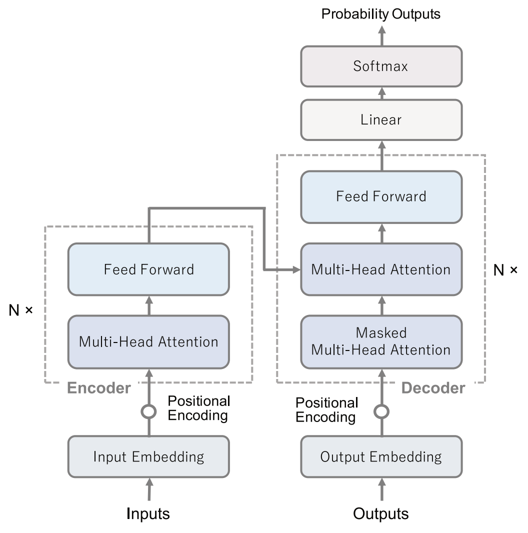


Fig. 3 Model architecture of Transformer.

3.3 Parallel Code Corpus

To build a more accurate translation model, a large parallel corpus is required in addition to a high-accuracy deep learning model, such as Transformer. However, the data resources of the parallel code corpus between a natural language and source code are insufficient compared to the parallel corpus between natural languages. The numbers of existing data resources for the parallel code corpus, such as Django (English–Python), CoNaLa (English–Python), and Euler corpus (Japanese–Python), are approximately 14,000, 3,000, and 700, respectively.

To overcome the shortage of parallel code corpora, several methods have been proposed, such as extracting comments and docstrings from the source code as bilingual text and extracting natural language descriptions from community sites such as Stack Overflow [5], [6]. However, these methods still have issues regarding the quality of the parallel code corpus, such as the risk of noise being included during the extraction. In addition, these methods use source code and natural language descriptions written by a third party, which may contain bugs in the source code and lead to different levels of abstraction in the expressions.

In recent years, benchmark datasets have been released to improve the deep learning models, such as CodeXGLUE [7], which collects the source code and comments from open sources. However, our goal is not to propose an improved deep learning model, but to create the coding assistance described in Section 2, which is our motivation for describing only the parts of the source code that cannot be remembered in natural language, and to convert those parts into the appropriate code. Therefore, we emphasize the importance of a parallel code corpus consistent with this motivation and avoid the use of CodeXGLUE.

4. Methods for Synthesizing the Parallel Code Corpus

We propose two methods for synthesizing a parallel code corpus based on back-translation (BT), a technique for generating bilingual translations by building a translation model that reverses the input and output of the designated model [8], [9], [10]. Our BT methods use the formal nature of the programming language and synthesize a bilingual translation based on the traditional compiler construction method instead of the BT model. In this section, we provide an overview of these two synthesis methods.

4.1 Top-Down Back-Translation

Top-down BT (TBT) is a BT method based on the classical compiler construction method. **Figure 4** illustrates the concept of the TBT. First, the Python code is converted into an AST based on syntactic parsing. Next, an AST is applied with predefined transformation rules (BT rules) in a top-down manner, and a pseudo-code in a natural language is generated. Finally, it is converted into a natural language description by adjusting the conjunctions.

The BT rules are given as a translation of the code corresponding to the node units of the AST as follows:

`if x:`	もし x のとき (If x, then)
`x % y`	x を y で割った余り (The remainder of x divided by y)
`x == y`	x が y に等しいかどうか (Whether x is equal to y)

The input source code is parsed into the AST, and the BT rules are applied in a top-down manner. At this point, BT rules are defined uniformly to always end with a noun, and thus when the final conjunction is adjusted, it can be connected as a Japanese sentence.

Synthesis Example (1)

`x % 2 == 0`	x を 2 で割った余りが 0 に等しいかどうか (Whether the remainder of x divided by 2 is equal to 0)

Synthesis Example (2)

`if x % 2 == 0:`	もし x を 2 で割った余りが 0 に等しいとき (If the remainder of x divided by 2 is equal to 0, then)

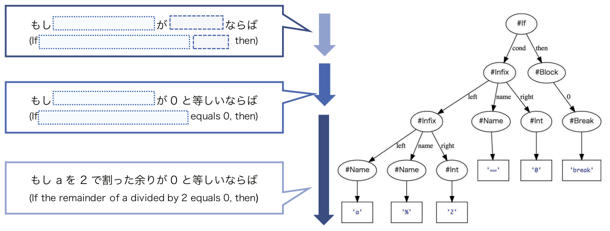
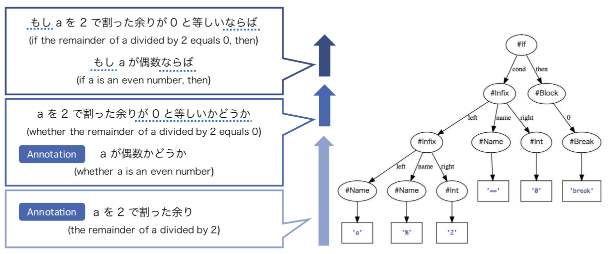


Fig. 4 Top-down back-translation.



もし a を 2 で割った余りが 0 と等しいならば
(if the remainder of a divided by 2 equals 0, then)
もし a が偶数ならば
(if a is an even number, then)
a を 2 で割った余りが 0 と等しいかどうか
(whether the remainder of a divided by 2 equals 0)
Annotation
a が偶数かどうか
(whether a is an even number)
Annotation
a を 2 で割った余り
(the remainder of a divided by 2)
#If
cond
then
#Infix
#Block
left
name
right
0
#Infix
#Name
#Int
#Break
#Name
#Name
#Int
'=='
'0'
'break'
'a'
'%'
'2'

Fig. 5 Bottom-up back-translation.

As the most significant advantage of TBT, it enables the construction of a parallel code corpus from a large amount of source code on the web, such as GitHub. However, TBT always outputs the same natural language description because the same BT rules are always applied when converting the code. Furthermore, TBT has a disadvantage in that, as the input code becomes longer, it results in unnatural sentences that people would not usually write, even if they are not grammatically incorrect.

4.2 Bottom-Up Back-Translation

Bottom-up BT (BBT) is a BT method that synthesizes bilingual translations in a bottom-up approach. **Figure 5** illustrates the concept of BBT.

In the BBT, we first manually prepare a parallel code corpus with multiple Japanese translations.

`x % n`
x を n で割った余り
(The remainder of x divided by n)

`x % 2 == 0`
x が偶数かどうか
(Whether x is an even number)
x は 2 で割り切れるかどうか
(Whether x is divisible by 2)

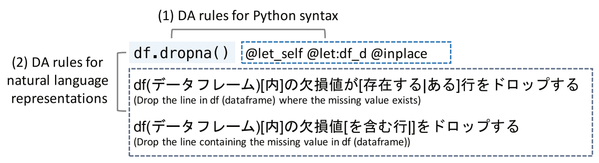
Unlike TBT, BBT provides bilingual translations to code fragments in expression units of arbitrary length, rather than in units of AST nodes. In this manner, we can obtain a more natural parallel code corpus.

Compared with natural languages, programming languages have only a limited variety of lexical patterns before and after an expression unit. Therefore, we assign lexical patterns that may appear before and after a prepared code fragment as `@not` annotation.

These annotations generate a parallel code corpus, as shown in **Table 1**. Thus, BBT synthesizes the parallel code corpus from a manually given code fragment in an expression unit in a bottom-up manner. However, the BBT does not synthesize the code at

Table 1 Example of extended logical expression.

	Python code	NL description
	`x % y == 0`	x を y で割った余りが 0 かどうか (Whether the remainder of x divided by y is zero)
@not	`not x % y == 0`	x を y で割った余りが 0 でないかどうか (Whether the remainder of x divided by y is non-zero)
@if	`if x % y == 0:`	もし x を y で割った余りが 0 ならば (If the remainder of x divided by y is zero, then)
@not @if	`if not x % y == 0:`	もし x を y で割った余りが 0 でないならば (If the remainder of x divided by y is non-zero, then)


(1) DA rules for Python syntax
(2) DA rules for natural language representations

```
df.dropna()  @let_self @let:df_d @inplace
```

df(データフレーム)[内]の欠損値が[存在する|ある]行をドロップする
(Drop the line in df (dataframe) where the missing value exists)

df(データフレーム)[内]の欠損値[を含む行|]をドロップする
(Drop the line containing the missing value in df (dataframe))

Fig. 6 Example of DA rule description.

the top of the tree structure. This is based on the analysis that the Transformer language model is a probabilistic model with lexical sequences and is significantly affected by the occurrence of the most recent lexical sequence.

In addition, we introduce a data augmentation (DA) notation to facilitate the definition of various natural language representations in the BBT.

- Synonym

$$\texttt{[表|データフレーム]} \begin{cases} \text{表 (Table)} \\ \text{データフレーム (Dataframe)} \end{cases}$$

- Word order reordering

$$\texttt{A を/B にする} \begin{cases} \text{A を B にする (Assign A to B)} \\ \text{B を A にする (Assign B to A)} \end{cases}$$

- Prefix/Suffix

$$\texttt{s(文字列)} \begin{cases} \text{文字列 s (String s)} \\ \text{s} \\ \text{s 文字列 (s string)} \end{cases}$$

The BBT can output an extended parallel code corpus by writing a few DA rules, as shown in **Fig. 6**.

5. Experiment Evaluation

We conducted experiments to determine the type of PyNMT model that can be built using the parallel code corpus constructed by the TBT and BBT. In this section, we summarize the experiments and their results.

5.1 Experiment Dataset

We prepared five different experiment datasets. **Table 2** summarizes the construction method and the number of datasets of the parallel code corpus. Euler [*1] is a dataset of the Python code used for solving the "Project Euler4" problem with manually annotated translations [11]. AOJ-T and AOJ-B are datasets constructed by collecting all Python code submitted to Aizu Online Judge [*2] by the top-100 highest scorers and applying TBT and BBT, respectively. Aizu Online Judge is different from the source code released on GitHub in that it has specific restrictions on the use of the library. Therefore, we can regard AOJ-T and AOJ-B as corpora of basic Python grammar and libraries. AOJ-T was constructed by applying TBT based on BT rules used to the collected Python code, and AOJ-B was constructed by adding DA rules to the BT rules.

DS is a dataset created by extracting APIs of commonly employed libraries for data analysis, such as Pandas and Matplotlib, from textbooks on data science and manually annotating them. DS-B is a dataset augmented by applying BBT to the DS. We confirmed that the number of datasets can be increased by a factor of 74 by applying the two DA rules proposed in the previous section. Although it takes time to write DA rules, the cost is minimal. This increase in the number of datasets also contributes to the improvement of the correct ratio, as will be shown in Section 5.3.

Table 2 Construction method and number of datasets of the parallel code corpus.

	Construction method	Number of datasets	Number of datasets (After removing duplicates)
Euler	Annotation	722	302
AOJ-T	TBT	250,673	27,684
AOJ-B	BBT	1,367	1,347
DS	Annotation	275	256
DS-B	BBT	20,407	20,175

5.2 Building Training Models

We built PyNMT models with five datasets divided into two parts, the training data and validation data, at a ratio of 7 : 3.

First, we preprocessed the datasets using a lexical analysis. We used Janome on the Japanese side and inserted spaces before and after the operators, parentheses, and quotation marks on the Python code side. In general preprocessing, we need to clean the datasets before conducting a lexical analysis; however, there is no need to clean the corpus synthesized by our proposed method because it does not contain noise such as that generated when extracting data from the web.

After a lexical analysis, we replaced the variable names and literals with special tokens, such as <A>. An example of this replacement is as follows.

Before replacement

Python :	`mylist.append('12345')`
Japanese :	`mylist に '12345' を追加する` (Append '12345' to mylist)

After replacement

Python :	`<A>.append(<B>)`
Japanese :	`<A> に <B> を追加する` (Append <B> to <A>)

With this replacement, the meanings of the variable names are no longer reflected during training; by contrast, the PyNMT model can replace variable names mechanically during a prediction. This replacement results in duplicates in the datasets, and we therefore removed these duplicates. The right side of Table 2 shows the number of datasets after the duplicate removal.

For training, we used Transformer implemented with PyTorch, applying the following parameters: 3 encoder and decoder layers, 8 heads, a batch size of 128, and 512 word embedding dimensions and hidden layer dimensions. We used cross-entropy as the loss function and Adam as the optimizer. The number of epochs in

*1 ahcweb01.naist.jp/pseudogen/
*2 judge.u-aizu.ac.jp/

Table 4 Experiment results (1): Example of prediction results.

	Input	Correct code	Predicted code	Correction
Euler	<A>の末尾に <B>を追加 (Append <B> to the end of <A>)	`<A> .  append ( <B> )`	`<A> <unk> <unk> <unk> <unk> <B> <unk>`	×
AOJ-T	<A>に<B>を掛けた値を<C>で割った値の浮動小数点数を出力する (Output the floating point number of <A> multiplied by <B> and divided by <C>)	`print(float(<A> * <B> / <C>))`	`print(float(<A> * <B> / <C>))`	✓
	入力された文字列を<B>で分割した字句列の各要素に整数を適用した列のリストを<A>とする (Assign a list of sequences in which the input string is divided by <B> and an integer is applied to each element of the lexical sequence to <A>)	`<A> = list(map(int, input().split(<B>)))`	`input = list(map(int, input().split(<B>)))`	×
AOJ-B	入力を半角スペースごとに整数としてリストで受け取り，<A>に代入する (Receive input as a list of integers per space and assign to <A>)	`<A> = list(map(int, input().split()))`	`<A> = list(map(int, input().split()))`	✓
	数列<A>の平均 (Average of sequence <A>)	`sum(<A>) / len(<A>)`	`len(<A>) / len(<A>)`	×
DS	<A>の行数・列数を確認 (Check the number of rows and columns in <A>)	`<A>.shape`	`<A>.shape`	✓
	重複している行を<A>から削除する (Remove duplicate lines from <A>)	`<A>.drop_duplicates()`	`from <A>.drop_duplicates()`	×
DS-B	データフレーム<A>の中の重複した行をドロップして，置き換える (Drop duplicate rows in dataframe <A> and replace them)	`<A> = <A>.drop_duplicates()`	`<A>.drop_duplicates(inplace=True)`	✓
	データフレーム<A>中にある<B>を未記入の値とする (Change <B> in dataframe <A> into an unfilled value)	`<A>.replace(<B>, np.nan)`	`replace <A>.replace(<B>, np.nan)`	×

the PyNMT model was set to terminate when the loss value of the validation data was minimized.

5.3 Experiment Result (1)

We evaluated the PyNMT model by preparing the test data using the holdout method to determine the type of Python code predicted. The test data were 1 : 1 split of the validation data. Herein, we present a quantitative evaluation based on two measures and a qualitative evaluation based on excerpts of the prediction results.

Table 3 summarizes the quantitative evaluation results. The correctness ratio is the ratio of the syntactically and interpretatively correct code predicted from the PyNMT model to the test data. When the number of test data was large, we conducted a statistical analysis through sampling. BLEU is a commonly used evaluation measure in machine translation that compares the predicted and correct sentences of a translation model and evaluates the translation accuracy based on the n-gram match ratio [12]. The score is 100 if the predicted sentence and the correct sentence are completely identical, and 0 if no match is found at all. We suppose that BLEU can be an indicator for programming languages that require strict grammatical descriptions because it can compare the correct code with the predicted code at the lexical level. However, because it cannot consider the syntax and interpretation, we calculated it as a reference value in this study.

As shown in the AOJ-T, AOJ-B, and DS-B experiments, the use of the synthetic method of the parallel code corpus proposed in this study shows a high correctness ratio. In particular, the parallel code corpus constructed using the BBT showed a higher correctness ratio.

Next, as a qualitative evaluation, we reviewed excerpts of the prediction results. **Table 4** summarizes the input sentences, correct code, predicted code, and correctness. Euler could not obtain the complete Python code because it output many special tokens `<unk>`, which indicates unknown words. Comparing AOJ-T and AOJ-B, AOJ-T shows that the input sentences of the test data are redundant. By contrast, AOJ-B has relatively natural Japanese sentences, although owing to its naturalness, we can see some cases in which the translation was incorrect because of a poor mapping with the tokens. Next, we compared DS and DS-B. DS often predicted unnecessary tokens at the beginning of the code, and most of the predicted code was incorrect, although it was not as poor as Euler's result. By contrast, DS-B predicted more correct code than DS. The predicted code shown as an example of the correct answer for DS-B is incorrect when compared only literally; however, we regard it as the correct code because the interpretation of a reassignment and `inplace=True` is the same.

Table 3 Experiment results (1): Correctness ratio and BLEU of the predicted code.

	Correctness ratio	BLEU
Euler	0.00%	5.63
AOJ-T	19.33%	75.54
AOJ-B	48.67%	78.84
DS	7.69%	31.80
DS-B	58.67%	69.76

5.4 Experiment Result (2)

As mentioned in the motivation section, our goal was to create a coding assistance system that converts the user's natural language input into Python code. Thus, we used sentences close to the user input as test data to evaluate whether the PyNMT model can predict the correct code when applied to a coding assistance system. We focused on the two models trained with AOJ-T and AOJ-B and compared two underlying synthesis methods. We did not construct DS-T, the corpus applying the top-down BT method to DS, because we did not have a source to obtain a large amount of Python code for applying TBT, and we could not cover all APIs; therefore, we compared only AOJ.

We used the correct answer rate of the prediction code through a manual evaluation and BLEU as evaluation indicators and calculated them using 150 cases of the same test data. As mentioned previously, we included BLEU as a reference value.

Table 5 summarizes the results of the quantitative evaluation. Comparing AOJ-T and AOJ-B, we can see that AOJ-B obtains a higher correctness ratio. However, unlike experiment result (1),

Table 6 Experiment results (2): Example of prediction results.

Input	Correct code	Predicted code (AOJ-T)	Correction (AOJ-T)	Predicted code (AOJ-B)	Correction (AOJ-B)
<A>の切り捨て (Truncation of <A>)	`math.floor(<A>)`	`math.floor(<A>)`	✓	`math.floor(<A>)`	✓
<A>が<B>よりも小さくなかったならば (If <A> is not smaller than <B>, then)	`if not <A> < <B> :`	`assert <A> < <B>`	×	`if not <A> < <B> :`	✓
<A>が<B>で割り切れないか (Whether <A> is divisible by <B>)	`<A> % <B> != 0`	`!= <A>.not <B>:`	×	`while <A> % <B> != 0`	×

Table 5 Experiment results (2): Correctness ratio and BLEU of the predicted code.

	Correctness ratio	BLEU
AOJ-T	5.33%	24.99
AOJ-B	28.00%	66.97

the correctness ratio of AOJ-T was 5.33%, which is a very small value.

Next, we qualitatively evaluated what types of predicted code was predicted. **Table 6** shows the predicted code and its correctness for each input sentence because we evaluated both AOJ-T and AOJ-B with the same test data. We can confirm that the predicted code of AOJ-T is significantly different from the correct code.

5.5 Findings and Perspectives

The synthesis of the parallel code corpus using TBT and BBT resulted in a higher correctness ratio than the small parallel code corpus constructed manually. TBT achieved the largest parallel code corpus, although the correctness ratio was higher for the PyNMT model trained with the corpus synthesized using BBT. One of the reasons why the correctness ratio decreased particularly for the user's input sentences is the unnaturalness of Japanese sentences when synthesized using TBT. TBT synthesizes Japanese sentences in a top-down manner, based on BT rules, resulting in mechanical and unnatural representations. However, BBT is based on manually defined translations, and thus it can predict the correct code for user-input sentences with a 28% correctness ratio. However, to use the PyNMT model for practical coding assistance, we must improve the correctness ratio.

6. Related Work

Code generation is a research area that has attracted significant interest in the software engineering field. Code generation generally refers to the task of transforming a natural language into a structural representation, such as a programming language or an AST. Improvements in the accuracy of the code generation will lead to new paradigms for no-code systems [13] that allow coding without memorizing programming languages' grammar rules.

Most previous studies on code generation have been based on a semantic analysis [14], which is an analysis technique of natural language processing. However, inspired by the recent development of deep learning technology and the expansion of its application range, researchers have proposed numerous approaches to applying deep learning technologies. This section provides an overview of recent studies on code generation in two major categories.

6.1 Semantic Analysis with Encoder–Decoder Model

The encoder–decoder model is a deep-learning model that transforms from one series length to another. One model, Seq2seq, was proposed in 2014 [15], [16] and has attracted significant attention in recent years. Transformer and Seq2seq have been used in a wide range of areas in natural language processing and have improved the accuracy of machine translation and question-answering systems.

In the field of software engineering, many attempts have been made to apply these models to tasks such as code generation and summarization. However, to apply the encoder–decoder model used in natural language processing to the source code, we face several problems arising from the difference in the nature of formal and natural languages. First, the output result of the encoder–decoder model is not formally guaranteed. To solve this problem, some studies have replaced the output text not with the code itself but with something structural such as a path representation of the AST [17]. Outputting the path leads to a solution that avoids situations in which the output code cannot be executed. Second, a simple encoder–decoder model cannot properly capture the structural information or context of a formal language. To address this problem, some studies have incorporated grammatical rules, tree structures, and contextual information into the model to effectively learn the code [18], [19].

6.2 Applying Source Code to Large-Scale Language Models

In recent years, many large-scale language models based on Transformer have been proposed in the natural language processing field and have been applied to source code [20], [21]. One technology that has surprised many engineers is GitHub's Copilot [22], which uses the GPT-3-based language model Codex. Codex enables the generation of function-by-function code by simply typing instructions to be executed [23].

It has also been applied to T5, which is a model that can realize multiple tasks simultaneously in a single model by solving all tasks in various natural language processing fields in a text-to-text format [24], as well as to GPT-2 and GPT-3. In the field of software engineering, T5 has been used to construct models that support multiple tasks related to coding [25] and models that generate different output units of source code from different input units of natural language [26].

The lack of training data is a major challenge in applying deep learning techniques. As a result, augmentation of training data has been a topic of growing interest in the natural language processing field, and several methods have been proposed, such as randomly replacing low-frequency words with other words [27], formulating the augmentation method as an optimization problem [28], and back-translation [8]. However, to our knowledge, no approach has been proposed to apply data augmentation to translation models for code generation. Our proposal differs from conventional data augmentation methods for natural languages in that it attempts to utilize the nature of source code, which is formal and can be transformed into syntax trees.

7. Conclusion

In this paper, we proposed two methods for synthesizing parallel code corpora, top-down BT and bottom-up BT, to overcome the problem of the shortage of parallel code corpora. We confirmed that these synthesis methods can be used to construct a more parallel code corpus than the existing one and build PyNMT models with a high correctness ratio. In particular, the PyNMT model trained using the parallel code corpus constructed through BBT can generate relatively correct code even from natural language expressions written by users. Therefore, we believe that BBT will become a core technology for achieving our goal of coding assistance with natural languages.

Future studies will involve improving the proposed method based on BBT to construct parallel code corpora containing expressions that are closer to the input sentences of the users. In addition, we would like to construct parallel code corpora more efficiently by incorporating various methods across disciplines, such as syntactic analysis techniques in language processing and distributed representations in natural language processing.

Acknowledgments We would like to thank Ms. Emi Nuijima (Softbank Corp.) for working with us from the conceptual stage of this study, and Ms. Shiho Takano (Japan Women's University) for her help in constructing the corpus.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I.: Attention Is All You Need, *CoRR*, Vol.abs/1706.03762 (2017) (online), available from ⟨http://arxiv.org/abs/1706.03762⟩.

[2] Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al.: Google's neural machine translation system: Bridging the gap between human and machine translation, arXiv preprint arXiv:1609.08144 (2016).

[3] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners, *OpenAI blog*, Vol.1, No.8, p.9 (2019).

[4] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners, arXiv preprint arXiv:2005.14165 (2020).

[5] Yin, P., Deng, B., Chen, E., Vasilescu, B. and Neubig, G.: Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow, *Proc. 15th International Conference on Mining Software Repositories*, *MSR '18*, pp.476–486, Association for Computing Machinery (online), DOI: 10.1145/3196398.3196408 (2018).

[6] Miceli Barone, A.V. and Sennrich, R.: A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation, *Proc. 8th International Joint Conference on Natural Language Processing* (*Volume 2: Short Papers*), pp.314–319, Asian Federation of Natural Language Processing (2017) (online), available from ⟨https://aclanthology.org/I17-2053⟩.

[7] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S. and Liu, S.: CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation, *CoRR*, Vol.abs/2102.04664 (2021).

[8] Edunov, S., Ott, M., Auli, M. and Grangier, D.: Understanding Back-Translation at Scale, *Proc. 2018 Conference on Empirical Methods in Natural Language Processing*, pp.489–500, Association for Computational Linguistics (online), DOI: 10.18653/v1/D18-1045 (2018).

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

[10] Ma, E.: NLP Augmentation (2019), available from ⟨https://github.com/makcedward/nlpaug⟩.

[11] Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T. and Nakamura, S.: Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation, ASE '1, pp.574–5845, IEEE Press (online), DOI: 10.1109/ASE.2015.36 (2015).

[12] Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J.: Bleu: A Method for Automatic Evaluation of Machine Translation, *Proc. 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Association for Computational Linguistics (online), DOI: 10.3115/1073083.1073135 (2002).

[13] Liao, W. and Guzide, O.: Low-Code/No-Code Software Development Platforms and Their Uses in Computer Science and Information Technology Education, *J. Comput. Sci. Coll.*, Vol.36, No.3, p.175 (2020).

[14] Lee, C., Gottschlich, J. and Roth, D.: Toward Code Generation: A Survey and Lessons from Semantic Parsing, arXiv preprint arXiv:2105.03317 (2021).

[15] Sutskever, I., Vinyals, O. and Le, Q.V.: Sequence to Sequence Learning with Neural Networks, *CoRR*, Vol.abs/1409.3215 (2014) (online), available from ⟨http://arxiv.org/abs/1409.3215⟩.

[16] Luong, T., Pham, H. and Manning, C.D.: Effective Approaches to Attention-based Neural Machine Translation, *Proc. 2015 Conference on Empirical Methods in Natural Language Processing*, pp.1412–1421, Association for Computational Linguistics (online), DOI: 10.18653/v1/D15-1166 (2015).

[17] Yin, P. and Neubig, G.: A Syntactic Neural Model for General-Purpose Code Generation, *Proc. 55th Annual Meeting of the Association for Computational Linguistics* (*Volume 1: Long Papers*), pp.440–450, Association for Computational Linguistics (online), DOI: 10.18653/v1/P17-1041 (2017).

[18] Iyer, S., Konstas, I., Cheung, A. and Zettlemoyer, L.: Mapping Language to Code in Programmatic Context, *Proc. 2018 Conference on Empirical Methods in Natural Language Processing*, pp.1643–1652, Association for Computational Linguistics (online), DOI: 10.18653/v1/D18-1192 (2018).

[19] Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L. and Zhang, L.: TreeGen: A Tree-Based Transformer Architecture for Code Generation, *Proc. AAAI Conference on Artificial Intelligence*, Vol.34, pp.8984–8991 (online), DOI: 10.1609/aaai.v34i05.6430 (2020).

[20] Perez, L., Ottens, L. and Viswanathan, S.: Automatic Code Generation using Pre-Trained Language Models, arXiv preprint arXiv:2102.10535 (2021).

[21] Svyatkovskiy, A., Deng, S.K., Fu, S. and Sundaresan, N.: Intellicode compose: Code generation using transformer, *Proc. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp.1433–1443 (2020).

[22] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I. and Zaremba, W.: Evaluating Large Language Models Trained on Code (2021).

[23] Phan, L., Tran, H., Le, D., Nguyen, H., Anibal, J., Peltekian, A. and Ye, Y.: CoTexT: Multi-task Learning with Code-Text Transformer, arXiv preprint arXiv:2105.08645 (2021).

[24] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. and Liu, P.J.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, *Journal of Machine Learning Research*, Vol.21, No.140, pp.1–67 (2020) (online), available from ⟨http://jmlr.org/papers/v21/20-074.html⟩.

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

[26] Clement, C.B., Drain, D., Timcheck, J., Svyatkovskiy, A. and Sundaresan, N.: PyMT5: multi-mode translation of natural language and Python code with transformers, arXiv preprint arXiv:2010.03150 (2020).

[27] Xie, Q., Dai, Z., Hovy, E., Luong, M.-T. and Le, Q.V.: Unsupervised Data Augmentation for Consistency Training, arXiv preprint arXiv:1904.12848 (2019).

[28] Wang, X., Pham, H., Dai, Z. and Neubig, G.: SwitchOut: An Efficient Data Augmentation Algorithm for Neural Machine Translation, *Proc. 2018 Conference on Empirical Methods in Natural Language Processing*, pp.856–861, Association for Computational Linguistics (online), DOI: 10.18653/v1/D18-1100 (2018).

Yuka Akinobu received her B.Sc. and M.Sc. from Japan Women's University in 2020 and 2022, respectively. She joined NTT Software Innovation Center, Japan. Her research interests include AI-supported software development, programming education, and natural language processing.

Teruno Kajiura received her B.Sc. from Japan Women's University in 2022. She is currently a master's student at Graduate School of Science, Japan Women's University. Her research interests include machine learning and natural language processing.

Momoka Obara received her B.Sc. from Japan Women's University in 2022. She is currently a master's student at Graduate School of Science, Japan Women's University. Her research interests include machine learning and programming education.

Kimio Kuramitsu is a Professor, leading the Software AI research group at Japan Women's University. His research interests include programming language, AI-supported software development, and programming education. He earned his B.E. at the University of Tokyo and his Ph.D. at the University of Tokyo under the supervision of Prof. Ken Sakamura.