mathematics

MDPI

Article

Pseudocode Generation from Source Code Using the BART Model

Anas Alokla [1], Walaa Gad [1], Waleed Nazih [2,*], Mustafa Aref [1] and Abdel-badeeh Salem [1]

[1] Faculty of Computers and Information Sciences, Ain Shams University, Abassia, Cairo 11566, Egypt
[2] College of Computer Engineering and Sciences, Prince Sattam Bin Abdulaziz University, Al Kharj 11942, Saudi Arabia
* Correspondence: w.nazeeh@psau.edu.sa

Abstract: In the software development process, more than one developer may work on developing the same program and bugs in the program may be fixed by a different developer; therefore, understanding the source code is an important issue. Pseudocode plays an important role in solving this problem, as it helps the developer to understand the source code. Recently, transformer-based pre-trained models achieved remarkable results in machine translation, which is similar to pseudocode generation. In this paper, we propose a novel automatic pseudocode generation from the source code based on a pre-trained Bidirectional and Auto-Regressive Transformer (BART) model. We fine-tuned two pre-trained BART models (i.e., large and base) using a dataset containing source code and its equivalent pseudocode. In addition, two benchmark datasets (i.e., Django and SPoC) were used to evaluate the proposed model. The proposed model based on the BART large model outperforms other state-of-the-art models in terms of BLEU measurement by 15% and 27% for Django and SPoC datasets, respectively.

Keywords: pseudocode generation; BERT; GPT; BART; natural language processing; neural machine translation

MSC: 68T50

Citation: Alokla, A.; Gad, W.; Nazih, W.; Aref, M.; Salem, A.-b. Pseudocode Generation from Source Code Using the BART Model. *Mathematics* **2022**, *10*, 3967. https://doi.org/10.3390/math10213967

Academic Editors: Nebojsa Bacanin and Catalin Stoean

Received: 2 September 2022
Accepted: 21 October 2022
Published: 25 October 2022

1. Introduction

In software development and maintenance, developers devote approximately 59% of their time to comprehending the source code [1,2]. Previous studies have shown that natural language description (i.e., pseudocode) is important for understanding the source code, which reduces the required time for software development. Writing a natural language description for every line of source code is challenging and difficult. As a result, developers often ignore this process although it is vital.

Several techniques were proposed for converting source code into pseudocode [3–6]. Previous work has used Neural Machine Translation (NMT) [7] and Statistical Machine Translation (SMT) [8,9]. SMT is based on utilizing statistical methods to find the alignment between input and output sentences, while NMT uses two neural networks (i.e., encoder and decoder) [10,11]. The training process of SMT is time consuming, and its results are not promising compared to NMT.

Transformer [12] was adapted to the Recurrent Neural Network (RNN) [13] and the Long Short-Term Memory (LSTM) [14] to overcome the vanishing gradient [15] and exploding gradient [16] problems.

There have been many transform-based models for language modeling such as Bidirectional Encoder Representations from Transformer (BERT) [17], Generative Pre-trained Transformer (GPT) [18,19], Bidirectional and Auto-Regressive Transformer (BART) [20], and Text-to-Text Transfer Transformer (T5) [21].

BERT and GPT are language models that were trained over a huge corpus of unlabeled text. The GPT is a unidirectional model in which training is used to predict the future left-to-right context, while BERT is based on bidirectional representations of unlabeled text by jointly conditioning the left and right context in all layers [17,18].

BART is a modified BERT with an emphasis on natural text generation. It consists of a bidirectional encoder and an auto-regressive decoder. Additionally, the training procedure begins by noisily perturbing the input text with functions such as deleting and masking before attempting to reconstruct the original text sequentially [20].

Furthermore, many applications such as Machine Translation (MT), Machine Summarization (MS), and Question Answering (QA) rely on BART.

In this paper, a novel adapted BART model was proposed for pseudocode generation. This model used a bidirectional transformer sequence for encoding as in the BERT model and an auto-regressive decoder for decoding as in the GPT model. This auto-regressive decoder generates a one-directional, left-to-right transformer sequence.

In addition, we formulated the pseudocode generation problem as MS because we train the adapted model using input data (i.e., source code) and output data (i.e., pseudocode) with the same languages (e.g., English language). The adapted model was trained with a source code and its equivalent pseudocode while in the testing phase; the model generated the pseudocode given the input source code. To evaluate the proposed model's performance, we measured its BLEU [13] measurement over two standard datasets (i.e., Django and SPoC).

To our knowledge, the proposed model is the first attempt to use BART models to generate pseudocode automatically. Our contributions include the following: (1) formulating pseudocode generation as an MS problem and using BART models for automatic pseudocode generation; (2) BART hyperparameter search optimization; (3) achieving BLEU measurements higher than other state-of-the-art models with 15% and 27% over Django and SPoC datasets, respectively.

This paper is organized as follows: Section 2 presents the related work, Section 3 presents the proposed model, Section 4 shows experimental results, and finally, Section 5 is the conclusion.

2. Related Work

Machine translation (MT) may be used for a variety of purposes in addition to linguistic translation from one language to another. Summarizing linguistic texts, locating descriptions of programming techniques, and translating Natural Language (NL) (pseudocode) into source code and vice versa are some of the tasks performed by MT.

In [3], the authors proposed a transformer-based system (DLBT) to convert the source code into pseudocode. This system starts with source code tokenization and embedding. In addition, it has a post-processing phase after transformer encoding and decoding to handle some minor errors that may occur during encoding. The biggest advantage of this system is the handling of missing tokens or those with low frequency.

In [4], the authors used retrieval mechanisms [22] to enhance DLBT's performance and solve the problem with input with missing or low-frequency tokens in the training dataset. This model has three steps; the first step retrieves sentences from the input that are the most similar to those in the training dataset. The second step passes the retrieved input into the DLBT. The third step achieves the targeted translation by the process of the replacement of the retrieved input with the corresponding retrieved translation from the training dataset.

In [5], the authors modified the transformer model for generating pseudocode from a source code. They used a transformer encoder containing two components: the first component is the normal encoder self-attention, while the second component uses a one-dimensional Convolutional Neural Network (CNN), then a Gated Linear Unit (GLU) [23], and a residual connection [24]. The output of the second component is responsible for

source code feature extraction. In addition, results from the encoder components are combined before feeding into the decoder.

In [6], the authors used a Long Short-Term Memory (LSTM) to build an NMT to generate the pseudocode from the source code. The encoder and decoder utilized LSTM and were combined through the attention layer. LSTM was used to avoid the vanishing gradient problem [15], while the attention layer aligned input and output sentences to improve the results. The limitation of this system was the time-consuming training process as it processes every sentence word by word.

In [25], the authors proposed a model consisting of three components to convert source code to pseudocode. The first component is an encoder with Bidirectional LSTM (BiLSTM), while the last component is an explanation decoder with attention and copy mechanisms [26]. In addition, a sketch decoder links the previously mentioned components. This decoder has LSTM and an attention layer.

In [9], the authors proposed a model based on Phrase-Based Machine Translation (PBMT) and Tree-to-String Machine Translation (T2SMT). The main role of PBMT is to align the input source code and output pseudocode. In addition, T2SMT converts the source code to a tree using Abstract Syntax Trees (ASTs) to maintain its context [27]. The disadvantage of this model is the low accuracy in the case of hitherto unseen sentences.

In [28], the authors proposed a Rule-Base Machine Translation system (RBMT) for converting the source code to pseudocode. The RBMT extracts more information from the source code and converts it to XML code using predefined templates. The proposed model is dataset based since RBMT should analyze the dataset to design the required rules and templates.

In [29], the authors designed a transformer-based model to convert the pseudocode to source code. The proposed model used BERT in the encoder, while the decoder was the traditional transformer decoder. In addition, the BERT was used to extract features from the input since the pseudocode is very similar to natural language. The model had two inputs in the training phase: the source sentence (i.e., pseudocode) to be fed into the encoder and the target sentence (i.e., source code) to be fed into the decoder. Later, in the translation phase, only one input was required, which is the source sentence.

Neural Machine Summarization (NMS) for summarizing source code to comments was proposed in [30]. This model deployed a retrieval-based mechanism to solve the problem of tokens having a low frequency or even those that did not exist in the training dataset. In addition, the model worked in two phases: the offline phase for training and the online phase for testing. In the offline phase, the model utilized an encoder, decoder, and attention layer to link them. In the online phase, an encoder with two levels of source retrieval was utilized; then, a decoder regenerated the output by fusing the two outputs of the aforementioned encoder. Furthermore, the proposed model was evaluated automatically and manually.

In [31], a large model for code representation was proposed to handle the limitations of the previous work such as the need for bilingual datasets for training. The proposed model has multi-layer transformer architecture similar to BART and T5. In addition, it used the same parameter settings as CodeBERT [32] and GraphCodeBERT [33]. The proposed model was pre-trained for both generation and classification tasks using large datasets of source code only. Furthermore, five downstream tasks were used to evaluate the proposed model.

In [34], the authors proposed a model for source code summarization. The proposed model was based on AST, multi-head attention, residual connection, and Graph Neural Networks (GNNs) [35]. Two encoders were utilized in this model: the first one processes the source code, while the second one takes AST embedding and feeds it into GNN. In addition, they utilized a decoder with three multi-head attention mechanisms. To generate the output, the summary tokens were fed into a decoder composed of six transformer decoding blocks. The proposed system was evaluated using two real Java and Python datasets.

3. The Proposed Model for Pseudocode Generation

In this paper, a novel model based on the bidirectional encoder and auto-regressive architecture was proposed for pseudocode generation. This model has two main components as shown in Figure 1: the bidirectional encoder as in the BERT model [17] and the auto-regressive decoder as in the GPT model [18].

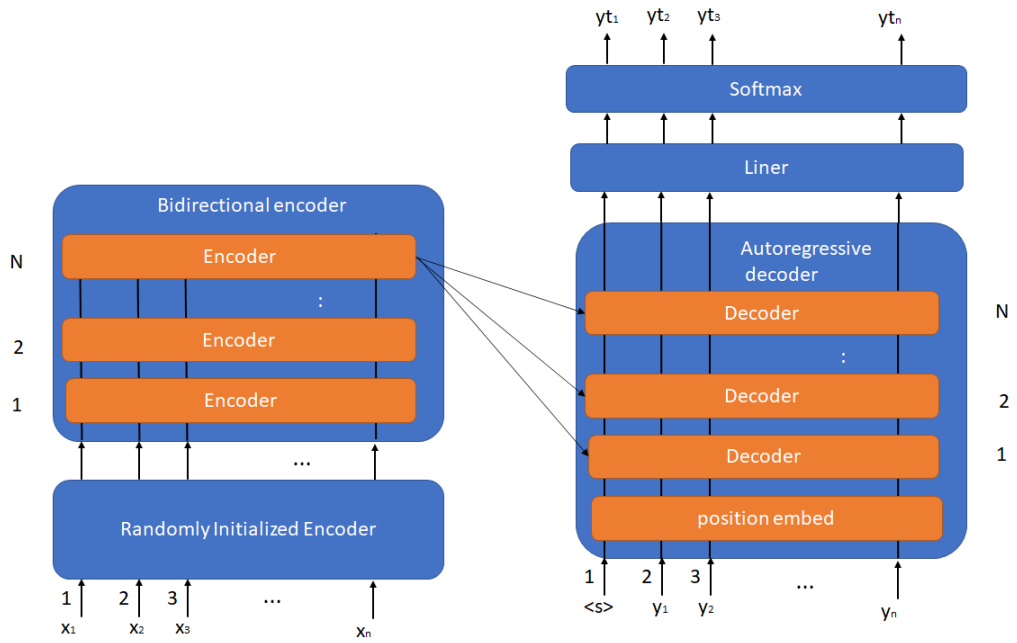


Figure 1. The BART model architecture.

In addition, there are other components such as a randomly initialized encoder for embedding the encoder sentences, the liner component for changing the output shape from auto-regressive to linear vectors, and the softmax component for predicting the linear vectors to tokens by using the softmax function.

3.1. Bidirectional Encoder

Figure 2 presents the bidirectional encoder of the BART model. The encoding process begins with tokens embedded as inputs for transformer encoder blocks (i.e., first layer). In addition, the output of the previous layer propagates to the next transformer layers. In the last layer, each transformer encoder outputs a vector that presents the features of an input token.

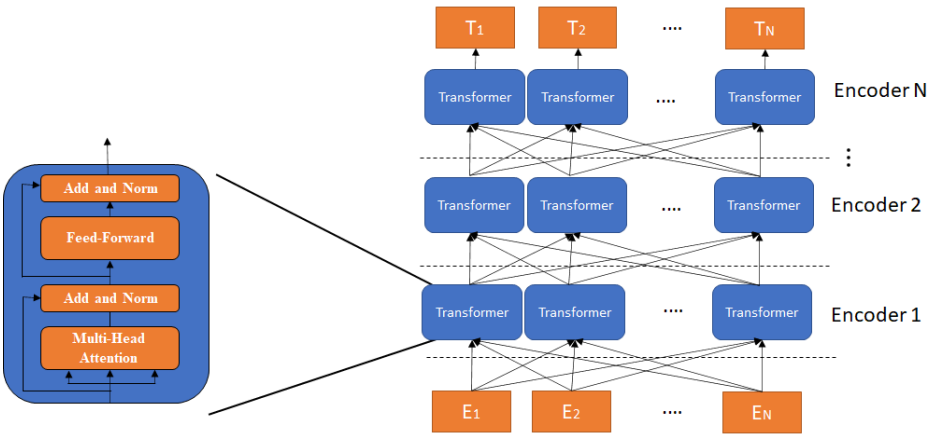


Figure 2. The bidirectional BART encoder architecture.

The transformer has three components: multi-head attention, add and norm, and feed-forward [12]. The multi-head attention has several attention layers running in parallel. The add and norm component adds the input vectors with the multi-head attention output and applies a normalized residual. The feed-forward is a feed-forward neural network.

3.2. Auto-Regressive Decoder

The auto-regressive decoder of the BART model is presented in Figure 3. The auto-regressive decoder takes tokens embedded as inputs. The output of each layer is processed for the next layer in all the transformer's layers. In the last layer, each transformer decoder generates a vector that represents the features of the output tokens.

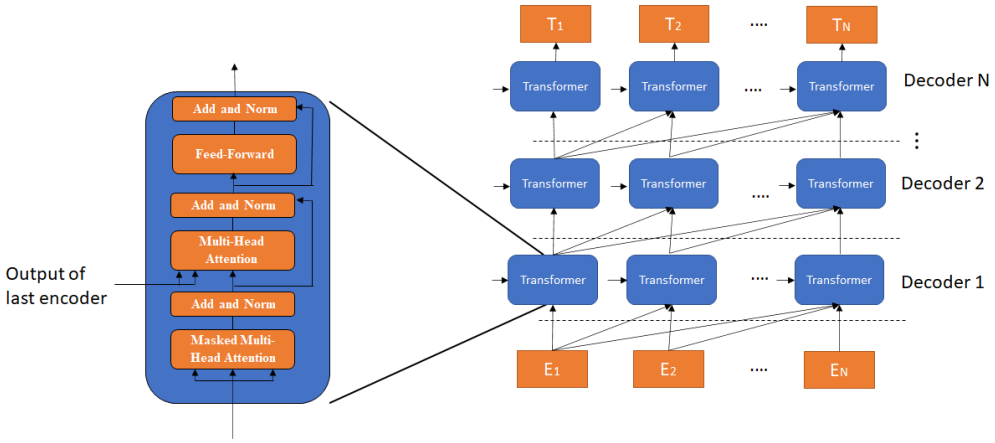


Figure 3. The auto-regressive BART decoder architecture.

The transformer component in the auto-regressive decoder is similar to that of the bidirectional encoder but with a second multi-head attention component to link the output of the bidirectional encoder with the auto-regressive decoder.

3.3. Pre-Trained BART Model

In our work, we fine-tuned two pre-trained BART models (i.e., large and base). The large model has 12 encoder and decoder layers. In addition, the attention layer in every transformer encoder and decoder has 16 attention heads. Furthermore, the dimension of layers (i.e., d_{model}) is 1024, and the total number of parameters of the model is 400 M.

The base model was pre-trained using a smaer dataset, so it has a smaller number of layers and parameters. Table 1 summarizes the differences between large and base BART models.

Table 1. Description of pre-trained BART models.

Model	n_{layers}	d_{model}	n_{heads}	n_{params}
BART large	12	1024	16	400 M
BART base	6	768	12	110 M

According to [12], transformer complexity for one layer is defined as $O(d \times n^2)$. However, the proposed model has 6 layers, then the total complexity is $O(6(d \times n^2) + 6(d \times n^2))$ for base BART, and $O(12(d \times n^2) + 12(d \times n^2))$ for large BART, where d is the embedding dimension or d_{model} and n is the number of words.

4. Experiments

Our proposed approach used the BART model and reported the performance over Django [6] and SPoC [36] datasets. In the first experiment, we tried the two versions of the

BART model (i.e., base and large). Since the large model achieved a better BLEU score over both datasets, we adopted it in the rest of the experiments.

In the second experiment, we fine-tuned the BART large model over Django [6] and SPoC [36] to achieve the best BLEU score. Finally, we compared between our proposed model's BLEU score and the reported score of other state-of-the-art models.

4.1. Datasets

Django [6] and SPoC [36] datasets were used to evaluate the proposed model. The Django dataset has Python source code, while the SPoC dataset has C++ source code. Datasets samples are shown in Figure 4.

Django Dataset

lines	Python code	Pseudocode
1	from threading import local	from threading import local into default name space.
2	import warnings	import module warnings.
3	from django . conf import settings	from django.conf import settings into default name space.
4	from django . core import signals	from django.core import signals into default name space.
⋮	⋮	⋮
18803	if contents is not None :	if contents is not None,
18804	self . characters (contents)	call the method self.characters with an argument contents.
18805	self . endElement (name)	call the method self.endElement with an argument name.

SPoC Dataset

lines	C++ code	Pseudocode
1	string s;	create string s
2	int x1, y1, x2, y2;	create integers x1, y1, x2, y2
3	cin >> s;	read s
4	x1 = s[0] - 96;	set x1 to s[0] - 96
5	y1 = s[1] - '0';	set y1 to s[1] - '0'
6	cin >> s;	read s
7	x2 = s[0] - 96;	set x2 to s[0] - 96
8	y2 = s[1] - '0';	set y2 to s[1] - '0'
⋮	⋮	⋮
197044	cout << v[0] << endl;	print the first element of v

Figure 4. Examples of Django and SPoC datasets.

The problem with the SPoC dataset is that some lines of code may have different pseudocode descriptions. This problem was solved in [3]. The Django and SPoC datasets were split into training, evaluation, and test datasets. Table 2 presents the number of samples in training, evaluation, and test datasets for Django and SPoC.

Table 2. The number of training, evaluation, and test samples in Django and SPoC datasets.

Dataset	Training	Evaluation	Test
Django	17,005	600	1200
SPoC	180,962	9000	15,183

4.2. Models' Parameters

The performance of the BART model is significantly influenced by its parameters. The performance might improve or worsen based on the values of these parameters.

In addition to the models' details mentioned in Table 1, we conducted discovery experiments and attempted different model parameters before settling on the following values: the learning rate, 0.001; the dropout, 0.1; the maximum length of the input sequence, 128 for the Django dataset and 64 for SPoC dataset; and the number of training epochs, 8 for Django dataset and 5 for SPoC dataset.

4.3. Performance Measures

The BLEU [13] metric was used for measuring the proposed model's accuracy. This metric measures the matching between the models' outputs and the actual outputs. The value of the BLEU metric ranges from zero to one. The highest value (i.e., one) indicates that we have a complete match between the models' outputs and the actual outputs. Different forms of BLEU were utilized based on different grams.

$$P_n = \frac{\sum_{ngram \epsilon C} count_{clip}(ngram)}{\sum_{ngram\prime \epsilon C} count(ngram\prime)}. \quad (1)$$

where P_n is the precision score based on n-gram (e.g., P_1 means the precision based on uni-gram). The $\sum_{ngram \epsilon C} count_{clip}(ngram)$ is the summation of the n-gram matches between the actual sentence and the predicted sentence.

In addition, the clipped n-gram is the number of predicted sentences. $\sum_{ngram\prime \epsilon C} count(ngram\prime)$. is the number of n-gramn the candidate sentence.

$$BP(r,c) = \left\{ \begin{matrix} e^{(1-\frac{|r|}{|c|})} \; if \; |r| \geq |c| \\ 1 \; otherwise \end{matrix} \right\}. \quad (2)$$

BP is the brevity penalty, r is the actual sentence, |r| is its length, c is the predicated sentence, and |c| is its length.

$$BLEU(r,c) = BP(r,c).\exp(\sum_{n=1}^{N} w_n.\log P_n) \quad (3)$$

Using n-grams up to length N and positive weights $\sum_{n=1}^{N} w_n$ summing to one where w_n is a vector such as (1,0,0,0) for uni-gram and (0.5,0.5,0,0) for bi-gram. The accuracy is the summation of all BLEU scores equal to one and multiplied by the number of sentences in the test data.

4.4. Results

In the first experiment, we compared the performance of different BART models over Django and SPoC datasets. An example of the pseudocode generated from the Python source code using the BART models is shown in Table 3. The pseudocode was produced in three different ways: manually by a skilled programmer; using BART base; and finally, using BART large.

In line 1, both BART base and BART large produce the appropriate pseudocode. In line 2, BART base does not generate "a" before "newline" and "character" after "new-line". It also replaces "into" with "to the". BART large does not generate "character" after "newline" and adds the "a" before the "self. stream". Both models generated correct semantic and grammatical sentences but were not as typical as the manual pseudocode.

In line 3, BART base adds "5" before the "arguments" while BART large generates the correct pseudocode. In line 4, both models removed "string" in the sentence, but BART base generates the first "string". Both models generate correct semantic and grammatical sentences but are not as typical as the manual pseudocode.

In line 5, BART base replaces the "add" with "sum", which is semantically correct, while the BART large model generates the correct pseudocode.

Table 3. The output of BART base and BART large for Django samples.

#	Python Code	Pseudocode
	Manually Generated Pseudocode	
1	if new_name! = col_name:	# if new_name is not equal to col_name,
2	self. stream. write (b'\n')	# write a newline character into self.stream file stream.
3	def __init__ (self, context, base, sysid, pubid):	# define the method __init__ with arguments self, context, base, sysid and pubid.
4	if path. startswith (('http://', 'https://', '/')):	# if path starts with string 'http://' or string 'https://' or with string '/',
5	dec_part = decimal_sep + dec_part	# add decimal_sep and dec_part, substitute the result for dec_part.
	BART Base	
1	if new_name ! = col_name :	# if new_name is not equal to col_name,
2	self. stream. write (b'\n')	# write newline to the self.stream file stream.
3	def __init__ (self, context, base, sysid, pubid):	# define the method __init__ with 5 arguments: self, context, base, sysid and pubid.
4	if path. startswith (('http://', 'https://', '/')):	# if path starts with string 'http://', 'https://' or '/',
5	dec_part = decimal_sep + dec_part	# sum decimal_sep and dec_part, substitute the result for dec_part.
	BART Large	
1	if new_name! = col_name:	# if new_name is not equal to col_name,
2	self. stream. write (b'\n')	# write a new line into a self.stream file stream.
3	def __init__ (self, context, base, sysid, pubid):	# define the method __init__ with arguments self, context, base, sysid and pubid.
4	if path. startswith (('http://', 'https://', '/')):	# if path starts with 'http://', 'https://' or '/',
5	dec_part = decimal_sep + dec_part	# add decimal_sep and dec_part, substitute the result for dec_part.

Table 4 shows the output of repeating the previous experiment over the SPoC dataset instead of the Django dataset.

Table 4. The output of BART base and BART large for SPoC samples.

	C++ Code	Pseudocode
	Manually Generated Pseudocode	
1	int len = s.size();	create integer len with len = size of s
2	cin >> s >> t;	read s and t
3	for (int j = 0; j < (n − i * 2 − 1)/2; j ++) {cout << "*";}	for j = 0 to (n − i * 2 − 1)/2, print *
4	if (a == "rock" and b == "paper") return −1;	return −1 if a is equal to "rock" and b = "paper"
5	for (int j = 1; j <= 2 * (i − 1) + 1; j ++) cout << "D";	for j = 1 to 2 * (i − 1) + 1 inclusive, print D
	BART Base	
1	int len = s.size();	create integer len with len = size of s
2	cin >> s >> t;	read s and t
3	for (int j = 0; j < (n − i * 2 − 1)/2; j ++) {cout << "*";}	for j = 0 to (n − i * 2 − 1)/2, exclusive
4	if (a == "rock" and b == "paper") return −1;	if a is equal to "rock" and b is equal"paper" return −1
5	for (int j = 1; j <= 2 * (i − 1) + 1; j ++) cout << "D";	for j = 1 to 2 * (i − 1) + 1, displayD
	BART Large	
1	int len = s.size();	create integer len with len = size of s
2	cin >> s >> t;	read s and t
3	for (int j = 0; j < (n − i * 2 − 1)/2; j ++) {cout << "*";}	for j = 0 to (n − i * 2 − 1)/2, print
4	if (a == "rock" and b == "paper") return −1;	if a is equal to "rock" and b is equal"paper" return −1
5	for (int j = 1; j <= 2 * (i − 1) + 1; j ++) cout << "D";	for j = 1 to 2 * (i − 1) + 1, print D

In lines 1 and 2, both BART base and BART large generate the correct pseudocode. In line 3, BART base adds the token "exclusive" instead of "print *"—maybe because a large number of for loops in the dataset have this token. Large BART was better since it generates "print" instead of "print *".

Both BART base and BART large generate logically and semantically correct pseudocode for lines 4 and 5. In line 4, a "return −1" was added after the condition, and "is equal" was generated instead of the "=" sign after "b". In addition, BART base generated "display" instead of "print" in line 5.

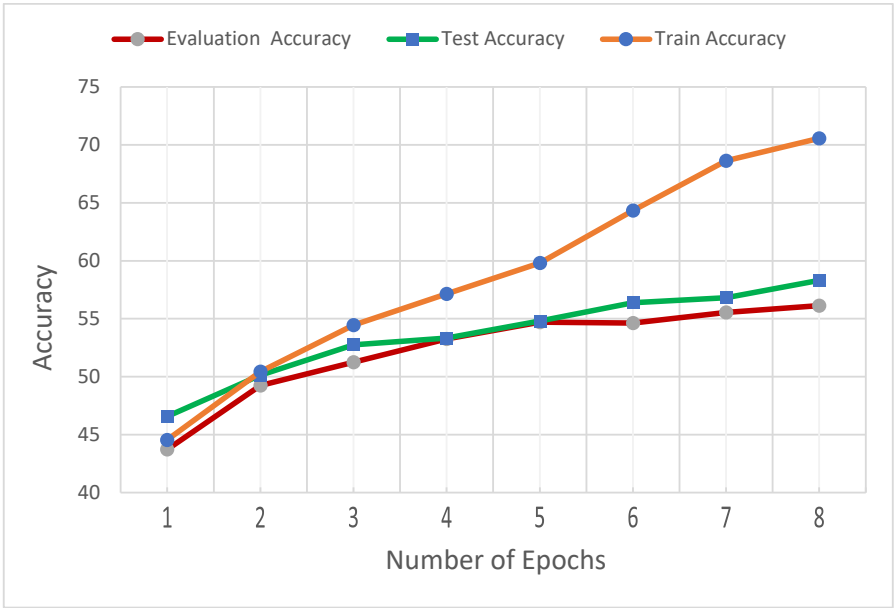
Since BART large achieved the better performance in the previous experiment, we conducted the second experiment to fine-tune it to achieve the best accuracy over the Django and SPoC datasets.

To evaluate the proposed model's performance, the value of loss function, BLEU metric, and accuracy were calculated for train, evaluation, and test datasets. Table 5 shows the aforementioned measurements over the output of fine-tuned BART large that was trained using the Django dataset and eight epochs.

Table 5. The performance of BART large using Django dataset and 8 epochs.

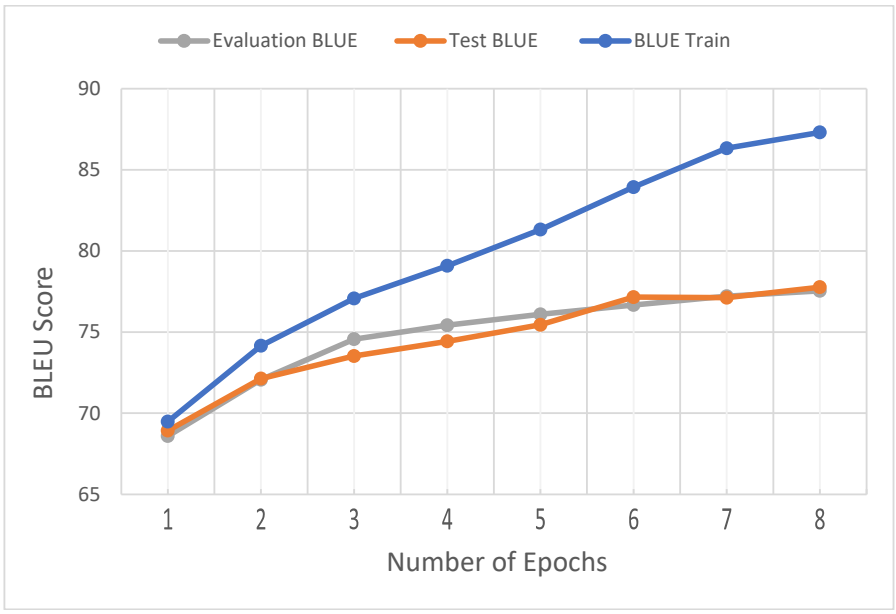
Epochs	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Train Loss	0.644	0.30	0.270	0.131	0.241	0.375	0.106	0.093
Evaluation Loss	0.543	0.470	0.435	0.417	0.417	0.426	0.437	0.441
Evaluation BLEU	68.59	72.06	74.56	75.41	76.09	76.66	77.20	77.53
Evaluation Accuracy	43.71	49.24	51.26	53.26	54.69	54.63	55.53	56.13
Test BLEU	68.92	72.12	73.52	74.42	75.44	77.15	77.11	77.76
Test Accuracy	46.57	50.12	52.75	53.33	54.80	56.39	56.80	58.31

In addition, Figure 5 shows (a) the accuracy of evaluation, test, and train datasets; (b) the BLEU of evaluation, test, and train datasets; and (c) the evaluation loss and training loss for the Django dataset.

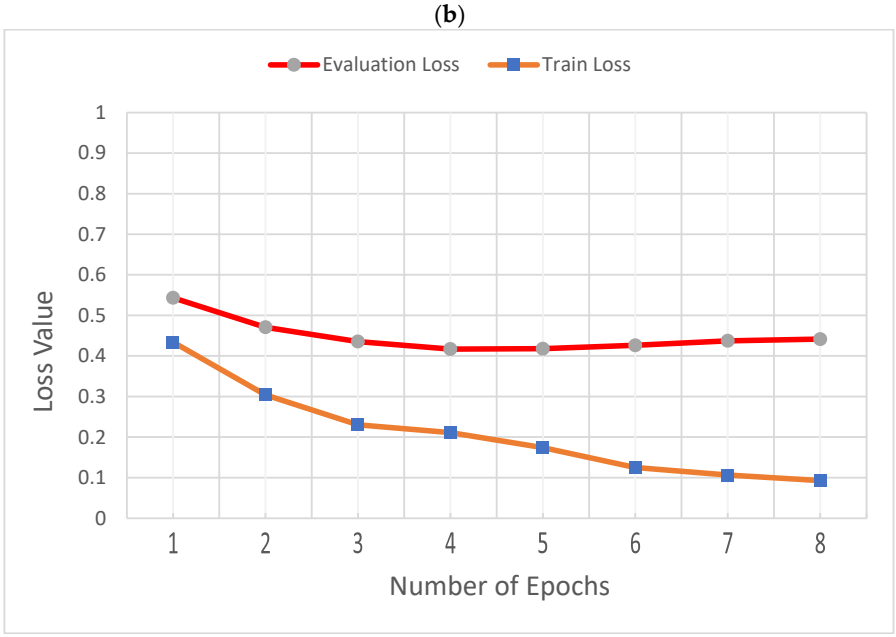


(**a**)

Figure 5. *Cont.*



(**b**)



(**c**)

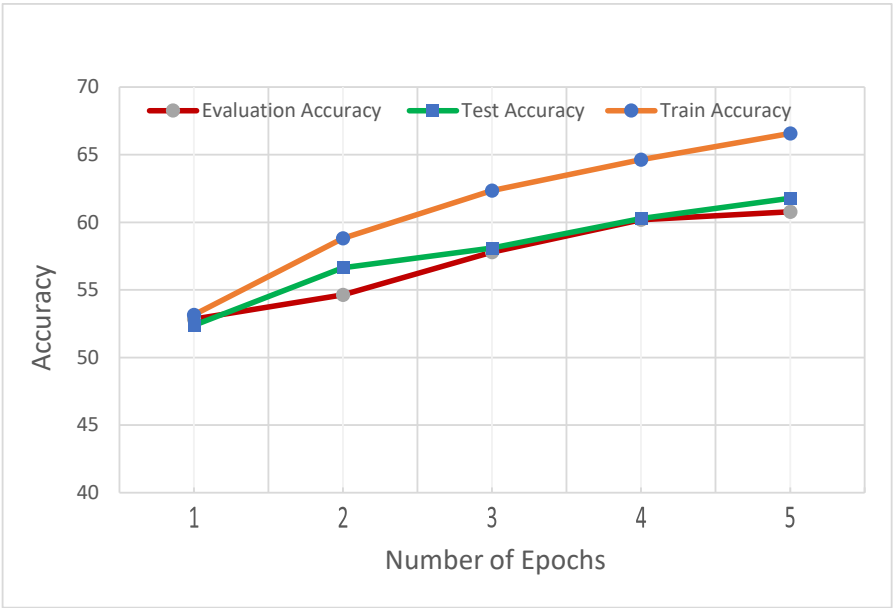
Figure 5. Performance of the BART large model for the Django dataset. (**a**) Accuracy of evaluation, test, and train datasets. (**b**) BLEU of evaluation, test, and train datasets. (**c**) Loss of evaluation and train dataset.

Table 6 shows the performance measurements over the output of fine-tuned BART large that was trained using the SPoC dataset and five epochs.

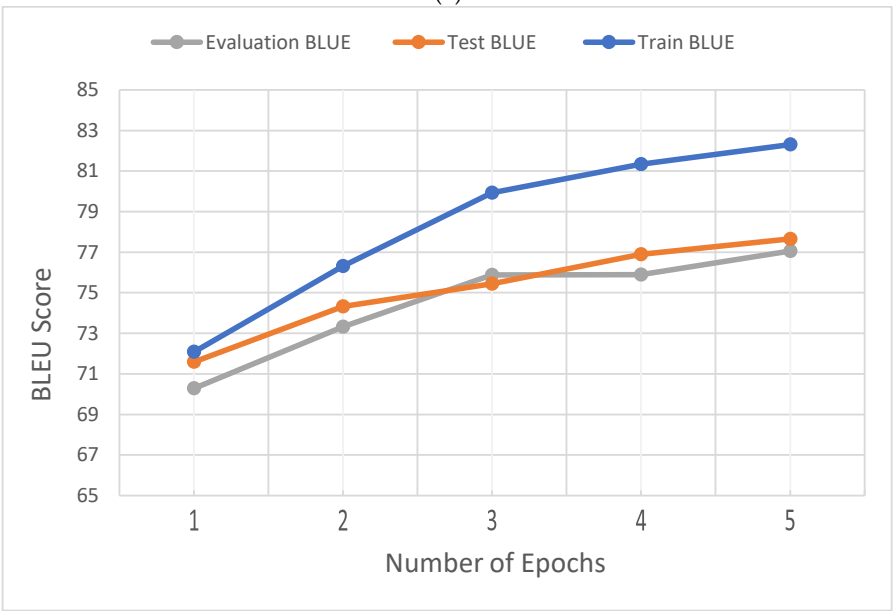
In addition, Figure 6 shows (a) the accuracy of evaluation, test, and train datasets; (b) the BLEU of evaluation, test, and train datasets; and (c) the evaluation loss and training loss for the SPoC dataset.

Table 6. The performance of BART large using SPoC dataset and 5 epochs.

Epoch	1	2	3	4	5
Train Loss	0.328	0.284	0.210	0.181	0.124
Evaluation Loss	0.443	0.400	0.375	0.357	0.327
Evaluation BLEU	70.28	73.32	75.87	75.89	77.05
Evaluation Accuracy	52.83	54.62	57.79	60.18	60.77
Test BLEU	71.58	74.32	75.43	76.89	77.65
Test Accuracy	52.37	56.62	58.09	60.28	61.77
Evaluation BLEU	70.28	73.32	75.87	75.89	77.05
Evaluation Accuracy	52.83	54.62	57.79	60.18	60.77

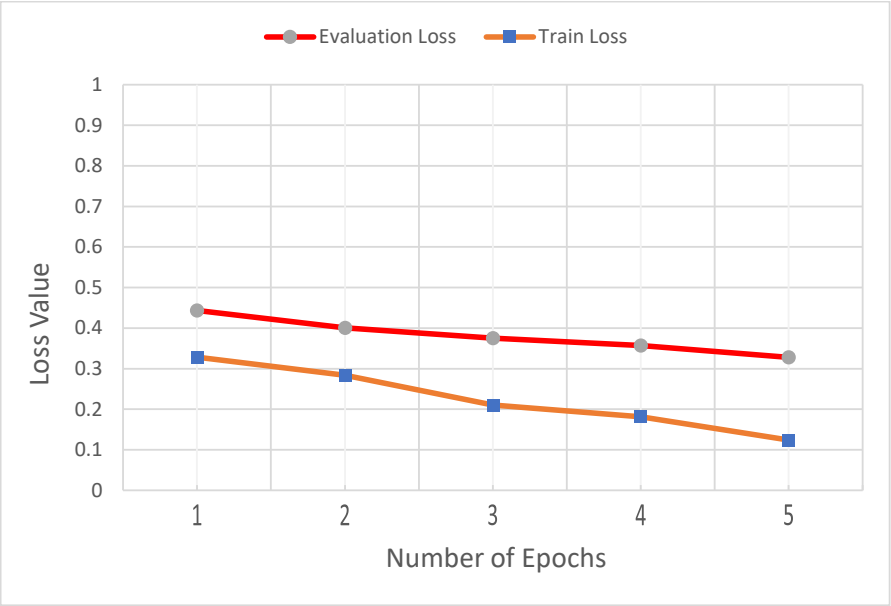


(**a**)



(**b**)

Figure 6. *Cont.*



(**c**)

Figure 6. Performance of BART large model for SPoC dataset. (**a**) Accuracy of evaluation, test, and train datasets. (**b**) BLEU of evaluation, test, and train datasets. (**c**) Loss of evaluation and train dataset.

Finally, a comparison between the proposed fine-tuned BART model and the state-of-the-art systems over the test dataset of Django and SPoC is introduced in Table 7. BART large achieved the best BLEU score over the Django and SPoC datasets.

Table 7. The comparison between the proposed model and state-of-the-art models on Django and SPoC datasets.

Dataset	Model	BLEU
Django	BART Large	**77.76**
	BART Base	**75.82**
	Levenshtein Retrieval on 6-layer DLBT [4]	61.96
	Levenshtein Retrieval on 8-layer DLBT [4]	61.29
	6-layer DLBT Not cross [3]	59.62
	8-layer DLBT Not cross [3]	58.58
	Code2NL [24]	56.54
	code2pseudocode [6]	54.78
	T2SMT [9]	54.08
	DeepPseudo [5]	50.817
	Code-GRU [25]	50.81
	Seq2Seq w Atten. [5]	43.96
	NoAtt [6]	43.55
	RBMT [28]	41.876
	CODE-NN [5,25]	40.51
	ConvS2S [5]	37.455
	Seq2Seq w/o Atten. [5]	36.483
	Seq2Seq [25]	28.26
	PBMT [9]	25.17
	SimpleRNN [6]	06.45
SPoC	BART Large	**77.65**
	BART Base	**76.26**
	Levenshtein Retrieval on 6-layer DLBT [4]	50.28
	6-layer DLBT Not cross [3]	48.12
	DeepPseudo [5]	46.454
	Transformer [5]	43.738
	Seq2Seq w Atten [5]	41.007
	ConvS2S [5]	34.197
	Seq2Seq w/o Atten. [5]	33.761
	CODE-NN [5,25]	32.105

5. Conclusions and Future Work

A novel, fine-tuned BART model was developed for automatic pseudocode generation. This model consists of two components: a bidirectional encoder as in the BERT model and a unidirectional decoder as in the GPT model. The proposed model was evaluated using benchmark datasets in Python and C++.

The model's results were better than those of other state-of-the-art models. Testing BART base with six layers and BART large with 12 layers over the Python dataset, they achieved 77.76% and 75.82% in terms of BLEU measure. In addition, BART base with six layers and BART large with 12 layers achieved 77.65% and 76.26% in terms of the BLEU measure over the C++ dataset.

We are planning to try other pre-trained models bigger than BART and having more parameters such as T5. Furthermore, we plan to revisit the fine-tuning of other models that were developed for programming languages such as CodeBERT.

Author Contributions: Conceptualization, W.G. and W.N.; methodology, W.G. and W.N.; software, A.A.; investigation, A.A., W.G. and W.N.; data curation, A.A.; writing—original draft preparation, A.A., W.G. and W.N.; writing—review and editing, W.G. and W.N.; supervision on main idea, references, figures, and experimental outputs, M.A. and A.-b.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Datasets used in experiments are public, and they were retrieved from the following URLs: https://ahcweb01.naist.jp/pseudogen/ and https://sumith1896.github.io/spoc/ (accessed on 10 July 2022).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Xia, X.; Bao, L.; Lo, D.; Xing, Z.; Hassan, A.E.; Li, S. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Softw. Eng.* **2017**, *44*, 951–976. [CrossRef]
2. Von Mayrhauser, A.; Vans, A.M. Program comprehension during software maintenance and evolution. *Computer* **1995**, *28*, 44–55. [CrossRef]
3. Gad, W.; Alokla, A.; Nazih, W.; Aref, M.; Salem, A.B. DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code. *CMC-Comput. Mater. Contin.* **2022**, *70*, 3117–3132. [CrossRef]
4. Alokla, A.; Gad, W.; Nazih, W.; Aref, M.; Salem, A.B. Retrieval-Based Transformer Pseudocode Generation. *Mathematics* **2022**, *10*, 604. [CrossRef]
5. Yang, G.; Zhou, Y.; Chen, X.; Yu, C. Fine-Grained Pseudo-Code Generation Method via Code Feature Extraction and Transformer. In Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021; IEEE: Manhattan, NY, USA, 2021.
6. Alhefdhi, A.; Dam, H.K.; Hata, H.; Ghose, A. Generating Pseudo-Code from Source Code using Deep Learning. In Proceedings of the 25th Australasian Software Engineering Conference (ASWEC), Adelaide, SA, Australia, 26–30 November 2018; IEEE: Manhattan, NY, USA, 2018; pp. 21–25.
7. Koehn, P. *Neural Machine Translation*; Cambridge University Press: Cambridge, UK, 2020.
8. Babhulgaonkar, A.; Bharad, S. Statistical Machine Translation. In Proceedings of the 1st International Conference on Intelligent Systems and Information Management (ICISIM), Aurangabad, India, 5–6 October 2017; IEEE: Manhattan, NY, USA, 2017; pp. 62–67.
9. Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; Nakamura, S. Learning to Generate Pseudo-Code from Source Code using Statistical Machine Translation. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; IEEE: Manhattan, NY, USA, 2015; pp. 574–584.
10. Sennrich, R.; Zhang, B. Revisiting Low-Resource Neural Machine Translation: A Case Study. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 211–221.
11. Mahata, S.K.; Mandal, S.; Das, D.; Bandyopadhyay, S. Smt vs. nmt: A comparison over hindi & Bengali simple sentences. *arXiv* **2018**, arXiv:1812.04898.
12. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5998–6008.

13. Reiter, E. A structured review of the validity of BLEU. *Comput. Linguist.* **2018**, *44*, 393–401. [CrossRef]
14. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
15. Roodschild, M.; Sardinas, J.G.; Will, A. A new approach for the vanishing gradient problem on sigmoid activation. *Prog. Artif. Intell.* **2020**, *9*, 351–360. [CrossRef]
16. Pascanu, R.; Mikolov, T.; Bengio, Y. Understanding the exploding gradient problem. *arXiv* **2012**, arXiv:1211.5063.
17. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
18. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving language understanding by generative pre-training. *Comput. Sci.* **2018**, preprint.
19. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. In Proceedings of the Advances in Neural Information Processing Systems 33 (NeurIPS 2020), Online, 6–12 December 2020; Volume 33, pp. 1877–1901.
20. Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; Zettlemoyer, L. 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, Online, 5–10 July 2020; pp. 7871–7880.
21. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67.
22. Zhang, J.; Utiyama, M.; Sumita, E.; Neubig, G.; Nakamura, S. Guiding Neural Machine Translation with Retrieved Translation Pieces. In Proceedings of the NAACL-HLT, New Orleans, LA, USA, 1–6 June 2018; pp. 1325–1335.
23. Dauphin, Y.N.; Fan, A.; Auli, M.; Grangier, D. Language Modeling with Gated Convolutional Networks. In Proceedings of the International Conference on Machine Learning, PMLR, Sydney, Australia, 6–11 August 2017; pp. 933–941.
24. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
25. Deng, Y.; Huang, H.; Chen, X.; Liu, Z.; Wu, S.; Xuan, J.; Li, Z. From Code to Natural Language: Type-Aware Sketch-Based Seq2Seq Learning. In Proceedings of the International Conference on Database Systems for Advanced Applications, Hyderabad, India, 11–14 April 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 352–368.
26. Gu, J.; Lu, Z.; Li, H.; Li, V.O. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, 7–12 August 2016; pp. 1631–1640.
27. Buch, L.; Andrzejak, A. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; IEEE: Manhattan, NY, USA, 2019; pp. 95–104.
28. Rai, S.; Gupta, A. Generation of Pseudo Code from the Python Source Code using Rule-Based Machine Translation. *arXiv* **2019**, arXiv:1906.06117.
29. Norouzi, S.; Tang, K.; Cao, Y. Code Generation from Natural Language with Less Prior and More Monolingual Data. *arXiv* **2021**, arXiv:2101.00259.
30. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Liu, X. Retrieval-Based Neural Source Code Summarization. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea, 5–11 October 2020; IEEE: Manhattan, NY, USA, 2020; pp. 1385–1397.
31. Niu, C.; Li, C.; Ng, V.; Ge, J.; Huang, L.; Luo, B. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 21–29 May 2022; IEEE: Manhattan, NY, USA, 2022; pp. 1–13.
32. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. Findings of the Association for Computational Linguistics: EMNLP 2020. *arXiv* **2020**, arXiv:2002.08155.
33. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
34. Guo, J.; Liu, J.; Wan, Y.; Li, L.; Zhou, P. Modeling Hierarchical Syntax Structure with Triplet Position for Source Code Summarization. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Dublin, Ireland, 22–27 May 2022; pp. 486–500.
35. Hamilton, W.; Ying, Z.; Leskovec, J. Inductive representation learning on large graphs. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; Volume 30.
36. Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; Liang, P.S. Spoc: Search-based pseudocode to code. In Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, BC, Canada, 8–14 December 2019; Volume 32.