BLOCSUM: Block Scope-based Source Code Summarization via Shared Block Representation

YunSeok Choi, Hyojun Kim, Jee-Hyong Lee
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
{ys.choi, rlagywns0213, john}@skku.edu

Abstract

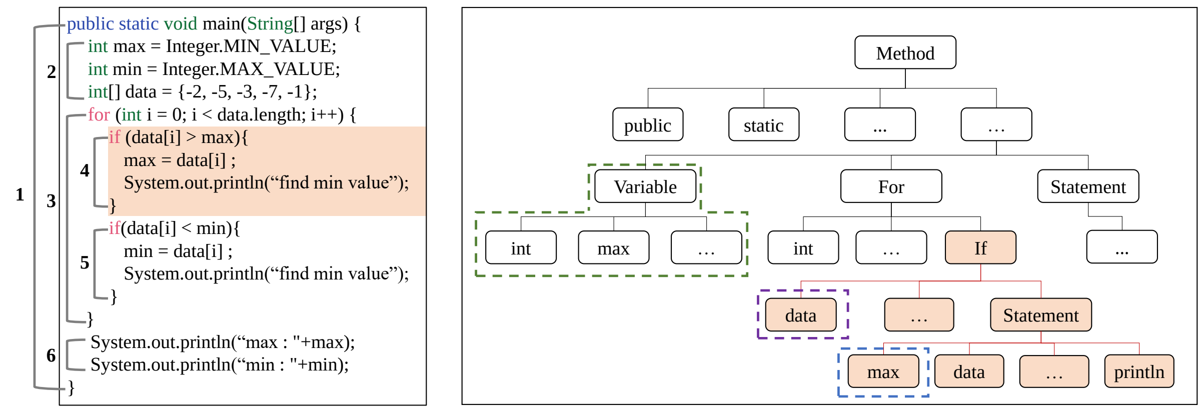
Code summarization, which aims to automatically generate natural language descriptions of the source code, has become an essential task in software development for better program understanding. Abstract Syntax Tree (AST), which represents the syntax structure of the source code, is helpful when utilized together with the sequence of code tokens to improve the quality of code summaries. Recent works on code summarization attempted to capture the sequential and structural information of the source code, but they considered less the property that source code consists of multiple code blocks. In this paper, we propose BLOCSUM, BLOck scope-based source Code SUMmarization via shared block representation that utilizes block-scope information by representing various structures of the code block. We propose a shared block position embedding to effectively represent the structure of code blocks and merge both code and AST. Furthermore, we develop variant ASTs to learn rich information such as block and global dependencies of the source code. To prove our approach, we perform experiments on two real-world datasets, the Java dataset and the Python dataset. We demonstrate the effectiveness of BLOCSUM through various experiments, including ablation studies and a human evaluation.

1 Introduction

A description of source code is very important in software development because it helps developers better understand programs. Advances in deep learning have enabled automatic code summarization and increased software maintenance efficiency. Previous approaches on automatic source code summarization can be categorized into sequence-based, structure-based, and hybrid approaches. Sequence-based approaches generated summaries by capturing the sequential information of source code (Iyer et al., 2016; Allamanis et al., 2016; Liang and Zhu, 2018; Hu et al., 2018b; Wei et al., 2019; Ye et al., 2020; Ahmad et al., 2020). They tokenized source code into a sequence of code tokens and encoded them using seq2seq models. Meanwhile, structure-based approaches used Abstract Syntax Tree (AST) to capture the structural information of code (Hu et al., 2018a; Fernandes et al., 2019; Shido et al., 2019; Harer et al., 2019; Zhang et al., 2019; LeClair et al., 2020; Liu et al., 2021; Lin et al., 2021; Allamanis et al., 2018; Wang and Li, 2021; Wu et al., 2021). They parsed the source code into the AST and utilized graph models such as Graph Neural Networks (GNNs). Some works flattened AST into a pre-order traversal sequence (Alon et al., 2019, 2018; LeClair et al., 2019; Wang and Li, 2021; Choi et al., 2021). Hybrid approaches utilized both the token sequence and the ASTs of codes (Wan et al., 2018; Wei et al., 2020; Zhang et al., 2020; Shi et al., 2021). They parallelly processed token sequences and ASTs with independent encoders and tried to merge them in the decoder.

However, the existing approaches have some limitations. Sequence-based approaches treated the source code as a single statement, so they incorporated only the sequence information of the code without any structural information such as code blocks. Structural information is very important to understand code because a snippet of code can be considered as a hierarchy of blocks.

Structure-based approaches tried to catch such structural information of code, but they considered less sequential information of code. A snippet of code is a sequence, so sequential information is also important to understand code. Another problem is that they depended only ASTs to capture structural information of code. However, ASTs are not suitable for capturing structural information because they are syntax trees for grammatical purpose. Since the AST is a tree, there is only one path between every pair of nodes. Any two nodes in ASTs are connected, but with a relatively long path, which hinders capturing the structural rela-



```
public static void main(String[] args) {
  int max = Integer.MIN_VALUE;
  int min = Integer.MAX_VALUE;
  int[] data = {-2, -5, -3, -7, -1};
  for (int i = 0; i < data.length; i++) {
    if (data[i] > max){
      max = data[i] ;
      System.out.println("find min value");
    }
    if(data[i] < min){
      min = data[i] ;
      System.out.println("find min value");
    }
  }
  System.out.println("max : "+max);
  System.out.println("min : "+min);
}
```

(a) Source code

(b) Abstract Syntax Tree

Figure 1: An example of code snippet and its AST in the Java language. (a) The orange box denotes the code block and the number means the index of the code block. (b) The nodes in the orange rectangular box belong to the same block in the AST.

tions of nodes. It causes difficulties in propagating structural information to distant nodes in the AST. Some structure-based approaches tried to alleviate these problems by providing additional graphs such as Control Flow Graph (CFG) and Program Dependence Graph (PDG), but the cost and time required to produce these graphs and integrate them into one graph are not negligible.

Hybrid approaches utilized both code and its AST. Since both sequential and structural information of code is necessary to understand code, the approaches showed higher performances than the previous ones. However, they failed to effectively merge two different types of information. They simply adopted independent encoders for each of them and tried to merge them in the decoder. Due to independent encoders, their representations are easy to be independent. It will make it hard to effectively merge both sequential information of code and structural information. Since the token sequence and the AST of code are just different descriptions of the same code, they need to be encoded so that they are correlated with each other.

To address these limitations mentioned above, we exploit the fact that the source code is a set of blocks consisting of multiple statements for a specific purpose. The code tokens in one code block are configured for the same purpose. As shown in Figure 1a, the code block `if { ... }` (orange) consists of statements that are executed when a certain condition is true. So, it needs to consider the information on which block each code token belongs to. To better capture structural information, we need to give each token not only positional information but also block positional information when encoding.

Since ASTs do not have enough information to capture the structural information of code, we need to modify ASTs. The additional information we try to add is block dependency and global dependency between nodes. For example, as shown in Figure 1b, the node "`max`" in the orange block of AST is only connected to its parent node, "`Statement`", but there exists implicit block dependency that the node "`max`" belongs to the same orange block as the node "`data`" in the purple dashed line. Furthermore, there exists implicit global dependency between nodes. For example, the node "`max`" in the orange box is the same variable as the nodes in the green dashed line. We need to add such information to ASTs.

Also, we need to match blocks in the token sequence with blocks in the AST. For example, code tokens ("`if`", "`(`", "`data`", `...`, "`}`") (orange) in Figure 1a can be mapped to its corresponding nodes in the AST ("`If`", "`data`", `...` , "`println`") (orange) in Figure 1b. Here, the block in the code and the block in the AST are the same part with an equal role, so we will utilize information on which block of code corresponds to which block of AST. Such information can make the code and AST correlated, and assist effective merging of two kinds of information.

In this paper, we propose BLOCSUM, BLOck scope-based source Code SUMmarization via shared block representation that utilizes block-scope information of token sequences and ASTs. First, we propose the shared block position em-

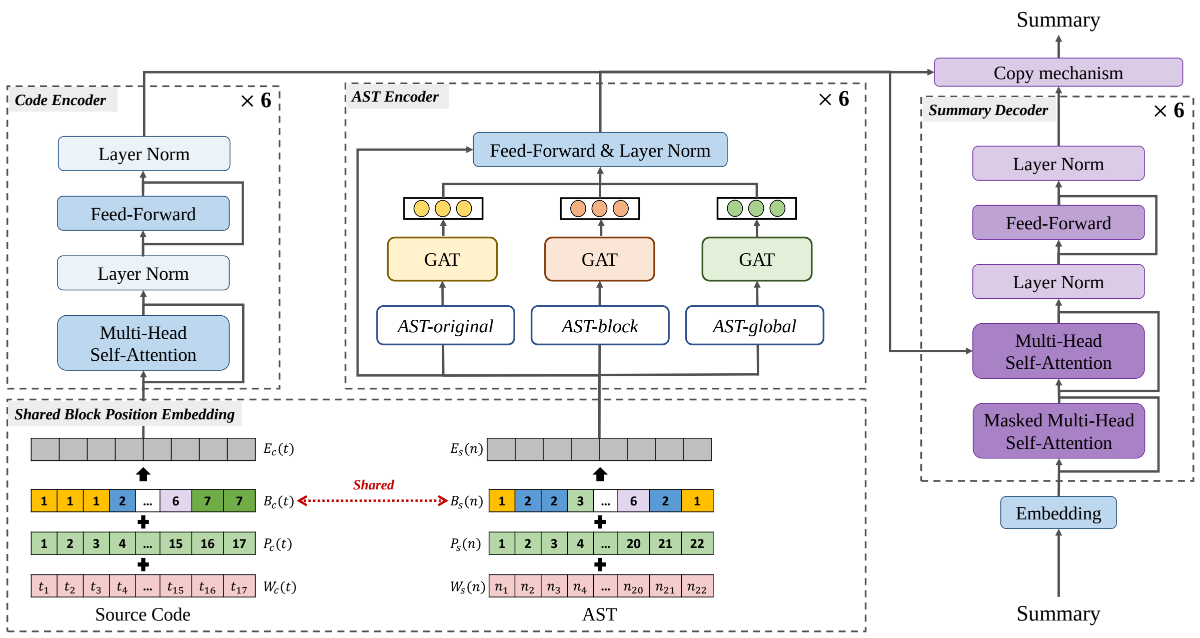


Figure 2: Overview of our proposed approach, BLOCSUM.

beddings for effectively representing the structure of the code block and combining a correlation between the code and the AST encoders. Furthermore, we develop simple yet effective variants of ASTs to learn rich information such as block and global dependencies of the source code. To validate our approach, we perform experiments on the Java dataset and the Python dataset. We prove the superiority of BLOCSUM through various experiments including ablation studies and a human evaluation.

2 BLOCSUM

In this section, we present the details of our model. Figure 2 shows the overall architecture of BLOCSUM. We first introduce the shared block position embedding and the abstract syntax tree variants and explain the architecture of BLOCSUM in detail.

2.1 Shared Block Position Embedding

We suppose that there are code tokens c_i in a code snippet $\mathbf{C} = \{c_1, c_2, ...\}$ and AST nodes n_i in its AST sequence $\mathbf{N} = \{n_1, n_2, ...\}$. We aim to predict a summary given the code tokens and the AST nodes.

Code blocks are the basic structural components of source code. Usually, code tokens in a block are gathered for a certain purpose, so the tokens need to be identified that they are in the same block. To distinguish which blocks are, we assign indexes to each block in the order of blocks in the code. Then each token has a block position with an index of the block it belongs to. If the code token is in nested blocks, we choose the innermost block index as the block position.

In order to utilize the block information of each token, we develop the block position embedding layer. Tokens in the same block have the same block position embedding. The code token embedding for the code encoder, E_c, is defined as follows:

$$E_c(t) = W_c(t) + P_c(t) + B_c(t) \tag{1}$$

for code token t. In the equation 1, W_c, P_c, and B_c are the word, position, and block position embedding layers for the code encoder, respectively. Two position embeddings, P_c and B_c, are learnable positional encoding.

We also combine the AST nodes with block position embeddings to ensure that nodes in the same block have identical block information when node representations are learned by the AST encoder. As with the block position of the code tokens, each node is assigned a block position value. Since the block in the AST is a sub-tree structure in the AST, the node has an index of the sub-tree to which it belongs. Nodes in the same block have the same block positional embedding as code tokens. The AST node embedding for the AST encoder, E_s, is defined as follows:

$$E_s(n) = W_s(n) + P_s(n) + B_s(n) \tag{2}$$

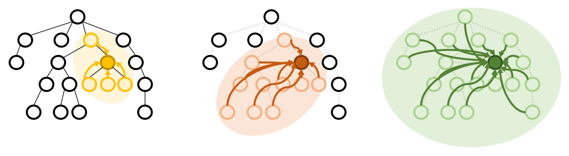


Figure 3: We introduce three different types of the AST: (a) ***AST-original*** referred as the original AST, (b) ***AST-block*** connected by edges between two nodes which are in the same code block, (c) ***AST-global*** connected by edges between all the nodes in the AST.

for node n. In Equation 2, W_s, P_s, and B_s are the word, position, and block position embedding layers for the AST encoder, respectively. The position of a node is defined as the position in the pre-order traversal sequence of the AST. Two position embeddings, P_s and B_s, are also learnable.

There are two different types of inputs, code and AST, but they are just different descriptions for the same snippet. If two encoders for code and AST learn representations for code tokens and AST nodes separately, their representations will be easy to be independent and very hard to effectively combine both sequential and structural information.

In order to correlate the representations learned by two encoders, we allow the encoders to share the block position embedding layer. If the code token and the AST node belong to the same block, they will have the same block position embedding value. That is, we utilize additional information on which parts of the code correspond to which parts of the AST to generate better representations. If the block position embedding layers are shared, the embeddings for a token, t, and a node, n are as follows:

$$E_c(t) = W_c(t) + P_c(t) + B(t) \tag{3}$$

$$E_s(n) = W_s(n) + P_s(n) + B(n) \tag{4}$$

where B is the shared block position embedding layer.

Shared block position embedding can effectively merge the information from two encoders. Also, it helps the code and the AST encoders to capture the structure of source code by providing block information.

2.2 Abstract Syntax Tree Variants

The original AST is a structure in which a node is connected only to its parent and children nodes. It contains local information, but it does not include the entire structure information of the code. Two nodes in the same block have implicit block dependency. There is also global dependency that two nodes have the same meaning even if their hop distance is very long. To utilize rich structural information such as block and global dependencies, we develop a simple yet effective method to reconstruct variants of AST. We define three variants of the AST: *AST-original*, *AST-block*, and *AST-global*.

AST-original is the original AST, which contains information on the local dependency between nodes, as shown in Figure 3a. The variants of AST are graphs of which nodes are the same as those of AST, but of which links are different.

AST-block is the first variant of AST. We obtain it by removing the edges in *AST-original*, and adding new edges between the nodes belonging to the same block to represent the block structure information, as shown in Figure 3b. It represents information on the block dependency between nodes in the AST.

AST-global is the second variant of AST. As shown in Figure 3c, we fully connect all the nodes in the AST. It represents the global and complete dependency between nodes in the AST. If the pre-order sequence of *AST-global* is learned by graph model, the node representations in the sequence represent context information of the AST.

Each of the three AST variants represents local, block, and global dependencies between nodes in the AST. When they are learned organically by the AST encoder, node representations will contain rich structural information of the AST.

2.3 BLOCSUM Architecture

Code Encoder Our code transformer encoder consists of 6 transformer layers (Vaswani et al., 2017). Each layer of the code transformer encoder is composed of two layers: multi-head self-attention (Vaswani et al., 2017) and feed-forward network. And residual connection (He et al., 2016) and layer normalization (Ba et al., 2016) are performed on each two sub-layers. The transformer encoder captures the sequential and block information of the code tokens to which the shared block position embedding is added.

AST Encoder We use Graph Attention Networks (GATs) (Velickovic et al., 2018) for learning three different AST variants defined above: *AST-original*, *AST-block*, and *AST-global*. Our AST encoder consists of 6 multiple GAT encoder layers. Each layer of the AST multiple GAT encoder consists of three GATs for each variant AST. Each GAT captures local dependencies for *AST-original*, block dependencies for *AST-block*, and global dependencies for *AST-global*, respectively.

For the l-th layer of the AST encoder, the process is performed as follows:

$$\begin{aligned} h_{ln}^{l} &= GAT_{ln}(A_{ln}, h_{n}^{l-1}) \\ h_{bn}^{l} &= GAT_{bn}(A_{bn}, h_{n}^{l-1}) \\ h_{gn}^{l} &= GAT_{gn}(A_{gn}, h_{n}^{l-1}) \end{aligned} \quad (5)$$

where GAT_{ln}, GAT_{bn}, GAT_{gn} and A_{ln}, A_{bn}, A_{gn} denote the GAT layers for three variant ASTs and the adjacency matrices in the *AST-original*, *AST-block*, and *AST-global*, respectively. Especially, the GAT for *AST-global* is the same as self-attention for learning the context of all nodes in the AST.

Finally, the three representations are combined and performed from residual connection and layer normalization by the following equation:

$$h_{n}^{l} = LN(h_{n}^{l-1} + FFN(h_{ln}^{l}, h_{bn}^{l}, h_{gn}^{l})) \quad (6)$$

where h_{n}^{l} is the concatenated node embedding in the l-th layer of AST GAT encoder, LN denotes layer normalization, and FFN is a feed-forward network.

With the deep AST encoder layers, the node representations combine and propagate the local, block, and global information of the AST.

Summary Decoder The summary transformer decoder consists of 6 transformer decoder layers (Vaswani et al., 2017). Code token representations are learned with sequence and block information in the code transformer encoder and node representations are learned with local, block, and global dependencies of the code in the AST encoder. Given the code and node representations learned from each encoder, the summary transformer decoder learns to predict the summary of the original code token by fusion of the code and the AST representations. The multi-head self-attention in the decoder is performed sequentially on the code representations and node representations.

Finally, when the summary transformer decoder predicts the t-th words, the copy mechanism (See et al., 2017) is applied to directly use the code tokens and AST nodes.

3 Experiment Results

3.1 Setup

Datasets We evaluate using the benchmarks of the two real-world datasets, the Java dataset (Hu et al., 2018b) and the Python dataset (Wan et al., 2018). The experiment datasets are divided into 69,708/8,714/8,714 and 55,538/18,505/18,502 for train/valid/test, respectively. For extracting the AST of each dataset, we used a java parser *javalang* in the Java dataset and a python parser *ast* in the Python dataset used by Wan et al. (2018). Refer to Appendix A for the statistics of the datasets in detail.

Hyper-parameters We set the maximum length of code, AST, and summary to be 200, 200, and 50, respectively. For training the model, we use Adam optimizer (Kingma and Ba, 2015). We set the mini-batch size as 80. The maximum training epoch is 100, and if the performance does not improve for 5 epochs, we stop early. Refer to Appendix C for the implementation details.

Evaluation Metrics We use BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004) as metrics. We adopt S-BLEU, which indicates the average sentence-level BLEU score. Refer to Appendix B in detail.

Baselines We adopt seq2seq models (Iyer et al., 2016; Hu et al., 2018a,b; Wei et al., 2019; Ahmad et al., 2020), graph2seq models (Eriguchi et al., 2016; Wan et al., 2018; Wu et al., 2021), hybrid models (Choi et al., 2021; Shi et al., 2021; Wu et al., 2021; Gong et al., 2022), and a pre-trained model CodeBERT (Feng et al., 2020) as the baselines. We fine-tuned the pre-trained model CodeBERT for code summarization.

3.2 Quantitative Result

Overall Result Table 1 shows the overall performance of BLOCSUM and baselines on the Java and Python benchmark datasets. First, we can see that BLOCSUM improves the performance by 4.47 and 2.64 BLEU, 5.68 and 4.66 METEOR, and 4.63 and 3.85 ROUGE-L on the Java and Python datasets compared to the sequence-based approach, TransRel. In comparison with the structure-based approach, SITTransformer, the performance of BLOCSUM improves by 3.29 and 1.05 BLEU, 4.53 and 2.11 METEOR, and 3.84 and 2.23 ROUGE-L on the two datasets. The result shows that it is effective to capture the overall structure of code when AST is utilized together with the sequence of code tokens. Moreover, BLOCSUM performs better than hybrid approaches. Compared to GCNTransformer, the result shows that it

Methods	Java			Python		
	BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
CODE-NN (Iyer et al., 2016)	27.60	12.61	41.10	17.36	09.29	37.81
Tree2Seq (Eriguchi et al., 2016)	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq (Wan et al., 2018)	38.22	22.75	51.91	19.28	09.75	39.34
DeepCom (Hu et al., 2018a)	39.75	23.06	52.67	20.78	09.98	37.35
TL-CodeSum (Hu et al., 2018b)	41.31	23.73	52.25	15.36	08.57	33.65
Dual Model (Wei et al., 2019)	42.39	25.77	53.61	21.80	11.14	39.45
Transformer (Ahmad et al., 2020)	44.58	26.43	54.76	32.52	19.77	46.73
CodeBERT* (Feng et al., 2020)	41.32	27.42	55.33	30.72	21.53	49.93
GCNTransformer (Choi et al., 2021)	45.49	27.17	54.82	32.82	20.12	46.81
SiTTransformer (Wu et al., 2021)	45.76	27.58	55.58	34.11	21.11	48.35
CAST (Shi et al., 2021)	45.19	27.88	55.08	-	-	-
SCRIPT (Gong et al., 2022)	46.89	28.48	56.69	34.00	20.84	48.15
BLOCSUM	**49.05**	**32.11**	**59.42**	**35.16**	**23.22**	**50.58**

Table 1: Comparison of our proposed model with the baseline models on the Java and Python datasets. We fine-tuned CodeBERT* with an input length of 200 and an output length of 50 for the two datasets.

is more effective to capture both the sequential and structural information of the code considering the local, block, and global dependencies of AST rather than a flattened AST. Also, BLOCSUM considering the correlation between code and node representations performs better than Tripletpos using two independent encoders for the code and AST. Finally, we compared our approach with CodeBERT, a strong pre-trained program language model. BLOCSUM performs significantly better than CodeBERT trained on large code data. The result shows that our approach is more appropriate for code modeling than the pre-trained model in the code summarization task.

Block Pos Emb	BLEU	METEOR	ROUGE-L
Java Dataset			
unuse	48.46	31.47	58.65
code block emb	48.53	31.56	58.63
ast block emb	48.54	31.77	58.92
separate	48.83	32.06	59.13
share	**49.05**	**32.11**	**59.42**
Python Dataset			
unuse	34.33	22.74	49.98
code block emb	34.50	22.84	50.10
ast block emb	34.56	22.83	50.08
separate	34.79	23.10	50.33
share	**35.16**	**23.22**	**50.58**

Table 2: Ablation study on shared block position embedding.

3.3 Qualitative Result

Ablation Study we perform ablation studies to validate the effectiveness of shared block position embedding and AST variants on the Java and Python datasets.

First, we design five models for comparison to verify the shared block position embedding: 1) not use block position embedding (*unuse*) 2) only use code block position embedding (*code block emb*) 3) only use AST block position embedding (*ast block emb*) 4) use separate block position embedding for code and AST (*separate*) 5) use shared block position embedding (*share*).

In Table 2, *code block emb* and *ast block emb* have better performance than *unuse*. This result shows that block position embedding is effective in capturing the block information in each encoder. Also when both code and AST encoder use each separate block position embedding, we can see that it is more effective than only using one block position embedding. Moreover, *share* has the best performance in comparison with other models. The shared block position embedding can learn the correlation between the code and AST encoders through the block-scope information rather than each separate block position embedding. Shared block position embedding not only effectively captures the structure of the code block, but also helps connect both the code and AST.

Second, we compare our approach with other combinations for verifying the effectiveness of variant ASTs: 1) *AST-original* (*o*) 2) *AST-block* (*b*), 3) *AST-global* (*g*).

As illustrated in Table 3, the results show that leveraging more structural information, such as block or global dependencies, performs better than modeling *AST-original* with only local dependency. Also, combining *AST-original*, *AST-block*, and *AST-global* has the best performance in comparison with

Combination	BLEU	METEOR	ROUGE-L
Java Dataset			
o	48.39	31.73	58.98
g	48.41	31.63	58.40
$o + g$	48.76	31.87	59.00
$o + b + g$	**49.05**	**32.11**	**59.42**
Python Dataset			
o	34.58	22.76	50.14
g	34.41	22.84	50.01
$o + g$	34.64	22.87	50.11
$o + b + g$	**35.16**	**23.22**	**50.58**

Table 3: Ablation study on combination of AST variants.

combining other combinations. We demonstrate that utilizing AST variants helps learn rich information such as block and global dependencies of the source code.

Human Evaluation We performed a human evaluation on the Python dataset to demonstrate the quality of generated summaries. We randomly select 100 code snippets and ask three people with knowledge of the Python language to evaluate the summaries. They are CS graduate students with many years of experience in Python languages. Following the human evaluation metrics of (Choi et al., 2021), we ask them to evaluate the 3 following metrics: 1) *Fluency* (Quality of the summary & grammatically correct), 2) *Relevance* (Selection of the consistent content in source code), 3) *Coverage* (Selection of the important content in source code). We show pairs of summaries generated from BLOCSUM and the baseline fine-tuned (Feng et al., 2020) to the evaluators, and they select one of win, tie, and loss in three metrics for both results.

Table 4 shows the results of human evaluation on the generated summaries on the Python dataset. The scores of fluency is lower, but the relevance and coverage are very higher than the baseline, CodeBERT. We analyzed the generated summaries of the two models and identified that BLOCSUM generates it similarly to the ground truth, reflecting the keyword of the code. CodeBERT, a pre-trained language model, can generate more fluent and grammatical summaries, but the length is relatively short and a very plain summary with no keywords. The average tokens in the ground truth are 10.14, while the average tokens in summaries generated by BLOCSUM and CodeBERT are 9.91 and 8.16, respectively. We think that short sentences are more grammatically advantageous than long sentences. BLOCSUM has the highest tie in terms

	Fluency	Relevance	Coverage
Win	76	143	125
Tie	114	64	80
Loss	110	93	95

Table 4: Human evaluation of the appropriateness of the generated adversarial example on the Python dataset.

of fluency but the highest win in terms of relevance and coverage. The result means that BLOCSUM reflects more the core characteristic of the code than CodeBERT.

Comparison with the baselines Table 5 shows the summary examples generated from our proposed model on the Python dataset. The result on the Python dataset example shows BLOCSUM predicts the keywords "`wsgi`" and "`request`" by reflecting block-scope information. Although there is no dependency between two words in the code and the original AST, BLOCSUM utilizes three different types of AST variants and jointly learns structural dependency in three aspects to improve the performance of the model. Refer to Appendix D for more summary examples on the Java and Python datasets.

4 Related Work

Sequence-based Approaches Iyer et al. (2016) and Allamanis et al. (2016) proposed to use Long Short Term Memory (LSTM) and Convolutional Neural Networks (CNNs) for the source code summarization. Liang and Zhu (2018) proposed a tree-based recursive neural network to represent the syntax tree of code. Hu et al. (2018b) and Chen et al. (2021) summarized the source code with the APIs knowledge. Wei et al. (2019) used a dual training framework by training code summarization and code generation tasks. Also, Ye et al. (2020) considered the probabilistic correlation between the two tasks. Choi et al. (2020) proposed attention-based keyword memory networks for code summarization. Ahmad et al. (2020) proposed a Transformer model using a relative position. However, these approaches have limitations in that they did not explicitly incorporate the structural information of the source code, which is just as crucial as capturing the code semantics. Also, they did not learn the code block information because they learned the code as a sequence of tokens.

Python Code

```
def simulate_request(app, method=’GET’, path=’/’, query_string=None, headers=None, body=None,
file_wrapper=None, params=None, params_csv=True):
    if (not path.startswith(’/’)):
        raise ValueError("path must start with ’/’")
    if (query_string and query_string.startswith(’?’)):
        raise ValueError("query_string should not start with ’?’")
    if (’?’ in path):
        raise ValueError(’path may not contain a query string. Please use the query_string parameter
                        instead.’)
    if (query_string is None):
        query_string = to_query_str(params, comma_delimited_lists=params_csv, prefix=False)
    env = helpers.create_environ(method=method, path=path, query_string=(query_string or ”),
                        headers=headers, body=body, file_wrapper=file_wrapper)
    srmock = StartResponseMock()
    validator = wsgiref.validate.validator(app)
    iterable = validator(env, srmock)
    result = Result(iterable, srmock.status, srmock.headers)
    return result
```

Ground Truth	simulate a request to a wsgi application .
CodeBERT	simulate a wsgi environment .
BLOCSUM	simulate a request to a wsgi request .

Table 5: A qualitative example on the Python dataset.

Structure-based Approaches Hu et al. (2018a) proposed an RNN-based model using the pre-order traversal sequence as input. Shido et al. (2019); Harer et al. (2019) adopted Tree-LSTM, Tree-Transformer to encode tree-based inputs. LeClair et al. (2020) proposed encoded AST using graph neural networks and trained LSTM. Liu et al. (2021) proposed a retrieval augmented method with Graph Neural Network (GNN). Zhang et al. (2019) proposed AST-based Neural Network (ASTNN) for encoding the subtree. Lin et al. (2021) proposed Tree-LSTM to represent the split AST for code summarization. Li et al. (2021) leverage the retrieve-and-edit framework to improve the performance for code summarization. Allamanis et al. (2018), Wang and Li (2021), and Wu et al. (2021) tried to capture rich information using additional graphs such as CFG and PDG. But, these approaches considered only the structural information of the AST without considering the sequential information of the code token.

Hybrid Approaches Alon et al. (2019) leveraged the unique syntactic structure of programming languages by sampling paths in the AST of a code snippet. LeClair et al. (2019) proposed ast-attendgru model that combines code with structure from AST Also, Choi et al. (2021) proposed a model that combines Graph Convolution Network and Transformer using AST. Wu et al. (2021) incorporated a multiview graph matrix into the transformer model. Shi et al. (2021) tried to hierarchically split and reconstruct ASTs using Recursive Neural Network for learning the representation of the complete AST. Wan et al. (2018) used a deep reinforcement learning framework to consider an AST structure and code snippets. Gong et al. (2022) proposed a structural position method to augment the structural correlations between code tokens. But, they encoded two types of representations independently without correlation and did not consider merging them.

Zhang et al. (2020) proposed a retrieval-based approach using syntactic and semantic similarity for source code summarization. Liu et al. (2021) proposed a hybrid GNN using a retrieval augmented graph method. Wei et al. (2020) proposed a comment generation framework using AST, similar code, and exemplar from code. Choi et al. (2023) proposed a self-attention network that adaptively learns the structural and sequential information of code. But, they tried to model the code using more code information through retrieval methods.

5 Conclusion

In this paper, we proposed BLOCSUM, BLOck scope-based source Code SUMmarization via shared block representation that utilizes block-scope information by representing various structures of the code block. We designed two methods using the fact that a code block is a fundamental structural component of the source code. We propose the first method, the shared block position embedding, for effectively representing the structure of the code block and merging a correlation between the code and the AST encoders. Furthermore, we developed to reconstruct simple yet effective AST variants to learn rich information such as block and global dependencies of the source code. Experimental results demonstrated the effectiveness of BLOCSUM and confirmed the importance of block-scope information in the code.

Limitations

In this paper, we conducted an experiment on code summarization using two benchmark datasets, the Java dataset (Hu et al., 2018b) and the Python dataset (Wan et al., 2018). BLOCSUM may need to be tested for its generalizability to other program languages. We chose two program languages (Java and Python) that were easily parsed to map the block position of Code and AST. We believe that since other programming languages have similar syntactic structures, BLOCSUM should be able to achieve similar performance on them as well.

Ethics Statement

This paper proposes block scope-based source code summarization via shared block representation that utilizes block-scope information by representing various structures of the code block, which is beneficial to increase the efficiency of developers. The research conducted in this paper will not cause any ethical issues or have any negative social effects. The data used is all publicly accessible and is commonly used by researchers as a benchmark for program and language generation tasks. Our proposed method does not introduce any ethical or social bias or worsen any existing bias in the data.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-00421, AI Graduate School Support Program(Sungkyunkwan University)) This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience Program(IITP-2023-2020-0-01821) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation)

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, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Miltiadis Allamanis, Hao Peng, and Charles 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. 2019. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning distributed representations of code.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *ArXiv preprint*, abs/1607.06450.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Fuxiang Chen, Mijung Kim, and Jaegul Choo. 2021. Novel natural language summarization of program code via leveraging multiple input representations. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2510–2520, Punta Cana, Dominican Republic. Association for Computational Linguistics.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2842–2851, Online. Association for Computational Linguistics.

YunSeok Choi, Suah Kim, and Jee-Hyong Lee. 2020. Source code summarization using attention-based keyword memory networks. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 564–570. IEEE.

YunSeok Choi, CheolWon Na, Hyojun Kim, and Jee-Hyong Lee. 2023. Readsum: Retrieval-augmented adaptive transformer for source code summarization. *IEEE Access*.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833, Berlin, Germany. Association for Computational Linguistics.

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

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. *arXiv preprint arXiv:2202.06521*.

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org.

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.

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

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. volume abs/2004.02843.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE.

Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–166. IEEE.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 5229–5236. AAAI Press.

Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 184–195. IEEE.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

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.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4053–4062, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. volume abs/1906.08094.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

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

Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 6559–6569.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1078–1090, Online. Association for Computational Linguistics.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 2309–2319. ACM / IW3C2.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE.

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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

A Statistics of Experiment Datasets

For obtaining ASTs of the Java and Python dataset, we use the *javalang*[1] and *ast*[2] library, respectively. Also, we tokenize the source code and the AST to subtokens as the form *CamelCase* and *snake-case* .

Dataset	**Java**	**Python**
Train	69,708	55,538
Valid	8,714	18,505
Test	8,714	18,502
Unique non-leaf nodes in ASTs	106	54
Unique leaf nodes in ASTs	57,372	101,229
Unique tokens in summaries	46,895	56,189
Avg. nodes in AST	131.72	104.11
Avg. tokens in summary	17.73	9.48

Table 6: Statistics of Java dataset (Hu et al., 2018b) and Python dataset (Wan et al., 2018). For obtaining their corresponding ASTs, we use the *javalang* and *ast* libraries, respectively.

B Evaluation Metrics

BLEU(Papineni et al., 2002) is a Bilingual Evaluation Understudy to measure the quality of generated code summaries. The formula for computing BLEU is as follows:

$$BLEU = BP \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n$$

where p_n is the geometric average of the modified n-gram precisions, ω_n is uniform weights $1/N$ and BP is the brevity penalty.

METEOR(Banerjee and Lavie, 2005) is used to measure how closely the metric scores match the human judgments about the quality of the translation. So unigram precision (P) and unigram recall (R) are computed and combined via a harmonic mean. The METEOR score is computed as follows:

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot P}$$

where $frag$ is the fragmentation fraction. α, β, and γ are three penalty parameters whose default values are 0.9, 3.0, and 0.5, respectively.

ROUGE-L(Lin, 2004) is used to apply Longest Common Subsequence in summarization evaluation. ROUGE-L used LCS-based F-measure to estimate the similarity between two summaries X of length m and Y of length n, assuming X is a reference summary sentence, and Y is a candidate summary sentence, as follows:

$$R_{lcs} = \frac{LCS(X,Y)}{m}, P_{lcs} = \frac{LCS(X,Y)}{n}$$

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$$

where $\beta = P_{lcs}/R_{lcs}$ and F_{lcs} is the value of ROUGE-L.

C Implementation Detail

We conducted experiments on Ubuntu 18.04 with 4 2080 Ti GPUs. The environment of the sever supports python 3.9, Cuda 10.2, pytorch 1.9, and pytorch geometric 1.7.

The average training and inference time for BLOCSUM takes about 40 and 0.5 hours, respectively. BLOCSUM has about 76 million parameters.

Hyper-parameter	**Size**
the maximum length of code tokens	200
the maximum length of AST nodes	200
the maximum length of summary tokens	50
Embedding dimension	512
The number of Code Encoder layers	6
The number of AST Encoder layers	6
The number of Code Decoder layers	6
Head of Attention	8
batch size	80
train epoch	100
early stop	5
learning rate	0.0005
learning decay	0.99
beam size	5

Table 7: Hyper-parameters of BLOCSUM.

[1]https://github.com/c2nes/javalang

[2]https://github.com/python/cpython/blob/master/Lib/ast.py

D Examples of Java and Python Datasets

Java Code

```java
@ Override public String toString() {
    String result;
    result=super.toString();
    if (m_CapabilitiesFilter != null) {
        initCapabilities();
        if (m_Capabilities != null) {
            if (m_Capabilities.supportsMaybe(m_CapabilitiesFilter) && !
                m_Capabilities.supports(m_CapabilitiesFilter)) {
                result="<html><font color=\"" + MAYBE_SUPPORT + "\">" + result + "</font></i><html>";
            }
    else if(!m_Capabilities.supports(m_CapabilitiesFilter)) {
                result="<html><font color=\"" + NO_SUPPORT + "\">"+ result+ "</font></i><html>";
            }}}
    return result;
}
```

Ground Truth	return a string representation of this tree node .
CodeBERT	build a string representation of the buff .
BLOCSUM	return a string representation of the capability .

Python Code

```python
def _get_codon_list(codonseq):
    full_rf_table = codonseq.get_full_rf_table()
    codon_lst = []
    for (i, k) in enumerate(full_rf_table):
        if isinstance(k, int):
            start = k
            try:
                end = int(full_rf_table[(i+1)])
            except IndexError:
                end = (start + 3)
            this_codon = str(codonseq[start:end])
            if (len(this_codon) == 3:
                codon_lst.append(this_codon)
            else:
                codon_lst.append(str(this_codon.ungap()))
    elif (str(codonseq[int(k):(int(k) + 3 )]) == ’—’):
        codon_lst.append(’—’)
    else:
        codon_lst.append(codonseq[int(k):(int(k) + 3)]))
    return codon_lst
```

Ground Truth	list of codon accord to full rf table for count .
CodeBERT	get cod ne .
BLOCSUM	get list that contain the codon in list .

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
Limitations

☑ A2. Did you discuss any potential risks of your work?
Limitations

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Abstract and 1. Introduction

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

3. Experiment Results

☑ B1. Did you cite the creators of artifacts you used?
3. Experiment Results

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
3. Experiment Results

☐ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Not applicable. Left blank.

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.

☐ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Not applicable. Left blank.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
3. Experiment Results and Appendix A

C ☑ Did you run computational experiments?

3. Experiment Results and Appendix C

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
3. Experiment Results and Appendix C

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☒ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
We used the same hyperparameter as the previous study.

☒ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
We adopted the median value among the 3 models.

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
3. Experiment Results and Appendix B, C

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?
Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.