Novel Natural Language Summarization of Program Code via Leveraging Multiple Input Representations

Fuxiang Chen
HKUST
fchenaa@cse.ust.hk

Mijung Kim
UNIST
mijungk@unist.ac.kr

Jaegul Choo
KAIST
jchoo@kaist.ac.kr

Abstract

The lack of description of a given program code acts as a big hurdle to those developers new to the code base for its understanding. To tackle this problem, previous work on code summarization, the task of automatically generating code description given a piece of code reported that an auxiliary learning model trained to produce API (Application Programming Interface) embeddings showed promising results when applied to a downstream, code summarization model. However, different codes having different summaries can have the same set of API sequences. If we train a model to generate summaries given an API sequence, the model will not be able to learn effectively. Nevertheless, we note that the API sequence can still be useful and has not been actively utilized. This work proposes a novel multi-task approach that simultaneously trains two similar tasks: 1) summarizing a given code (code to summary), and 2) summarizing a given API sequence (API sequence to summary). We propose a novel code-level encoder based on BERT capable of expressing the semantics of code, and obtain representations for every line of code. Our work is the first code summarization work that utilizes a natural language-based contextual pretrained language model in its encoder. We evaluate our approach using two common datasets (Java and Python) that have been widely used in previous studies. Our experimental results show that our multi-task approach improves over the baselines and achieves the new state-of-the-art.

1 Introduction

Developers spend the most time writing code but not much in writing its description. It is reported that much of the developers' code does not have any description (Hu et al., 2018b). This has detrimental effects on other developers who will be reading and trying to understand the code base

```
/**
* Returns area by name
*/
public int getAreaByName(String name) {
    return getArea(name);
}
```

(a) Code 1 having API Sequence `getArea`

```
/**
* Returns twice the area given a name
 */
public int getTwiceAreaByName(String buildingName) {
     return getArea(buildingName) * 2;
}
```

(b) Code 2 having API Sequence `getArea`

Figure 1: Code 1 and Code 2 implement different functionalities but use the same API sequence.

(Wei et al., 2019). To alleviate effort in writing the code description, code summarization, the task of automatically generating code description given a piece of code, has been proposed in the software engineering and AI community (Haiduc et al., 2010; Moreno et al., 2013; Iyer et al., 2016; Hu et al., 2018a).

Previous work on code summarization using an auxiliary model trained to produce API (Application Programming Interface) embeddings showed promising results when applied to a separate code summarization model (Hu et al., 2018b). However, different code may assume the same set of API sequence. For example, Figure 1a and 1b show two different code snippets having the same API sequence, *getArea*. Despite having the same API sequence, the code summary shown in the comments on top is different: Figure 1a is about getting an area given a name, while Figure 1b is about doubling an area. Thus, training a model to summarize the given code based on its API sequence may induce confusion into the model. Nevertheless, we note that the API sequence can still be useful and has not been actively explored.

In this work, we further leverage the API sequence for code summarization. Specifically, we propose a novel multi-task approach that simultaneously trains two similar tasks: 1) code to summary, and 2) API sequence to summary. Our model consists of an encoder-decoder architecture. That is, we propose a novel code-level encoder based on BERT (Devlin et al., 2019), which has recently shown remarkable improvement in numerous NLP downstream tasks. Our encoder is able to express the semantics of code and obtain modeling for every line of code. Our work is also the first code summarization work that utilizes a natural language-based contextual pre-trained language model in its encoder. For multi-task learning, the two different tasks utilize the same set of shared layers that produce contextual embeddings to further train the individual tasks.

We evaluate our approach on two popular datasets (Java and Python) that have been widely used in previous studies. Our experimental results show that by learning to identify lines of code, our model is able to learn more effectively. Furthermore, our multi-task approach improves over the baselines and achieves the new state-of-the-art performance.

In summary, our key contributions include:

- A novel multi-task learning model that consists of two different but semantically similar tasks of generating summaries from either code and API sequence.
- A novel approach of representing lines of code that leads to improved performance.
- Experimental results compared with baselines that achieve state-of-the-art performance in code summarization.

2 Related Work

Most existing approaches that perform code summarization using neural networks define the output task to be a sequence generation (Iyer et al., 2016; Hu et al., 2018a,b; Liang and Zhu, 2018). These approaches leverage recurrent encoder-decoder models with attention mechanisms. One prior work proposed a new convolutional attention model for code summarization that outputs short name-like summaries (Allamanis et al., 2016). Different from this previous work, our work is a multi-task (Code to Summary and API Sequence to Summary) approach that utilizes a contextual pre-trained language model in the area of code summarization.

There has been research that leverages source code representation for code summarization. A number of recent work transforms the source code into Abstract Syntax Tree (AST) and encodes it using TreeTransformer (Harer et al., 2019), Tree-LSTM (Shido et al., 2019), and Graph Neural Networks (LeClair et al., 2020). Another prior work has improved the code summarization using the AST by flattening it into a sequence through structure-based traversal (Hu et al., 2018a). Later work further improved the model by proposing a representation that decouples the code structure from structure-based traversal (LeClair et al., 2019). Our work leverages a more readable and simplified structure than AST by tokenizing every line of code (e.g., `[CLS]int i = 0[SEP]`).

Other existing work leverages various learning techniques such as reinforcement learning (Wan et al., 2018), dual learning (Wei et al., 2019), and retrieval-based techniques (Zhang et al., 2020) to build the code summarization models. Recent work leverages a transformer model to generate natural language summary (Ahmad et al., 2020). In contrast, our work uses a pre-trained language model as a code-level encoder.

A previous technique uses API usage information that enhances the code summarization model, showing the effectiveness of the knowledge from API sequence (Hu et al., 2018b). We noted the exploration of API sequence in code summarization is limited. Although multiple different codes can have the same API sequence and the API sequence may not contain the full information of the code, the API sequence can directly provide more structural information of the code, e.g., the data type used, and their sequences. This can be viewed as a friendlier/structured/cleaner intermediate representation. Though it may be lossy in the code, such intermediate representation facilitates the model training as we have empirically shown that it is improving the model performance. Contrary to (Hu et al., 2018b), our work consists of a multi-task approach that summarizes code as well as API sequence. Our experiments show that this multi-task approach is useful in learning a better code-to-summary model.

There has been work that uses a dual task model (Wei et al., 2019) in which one model is

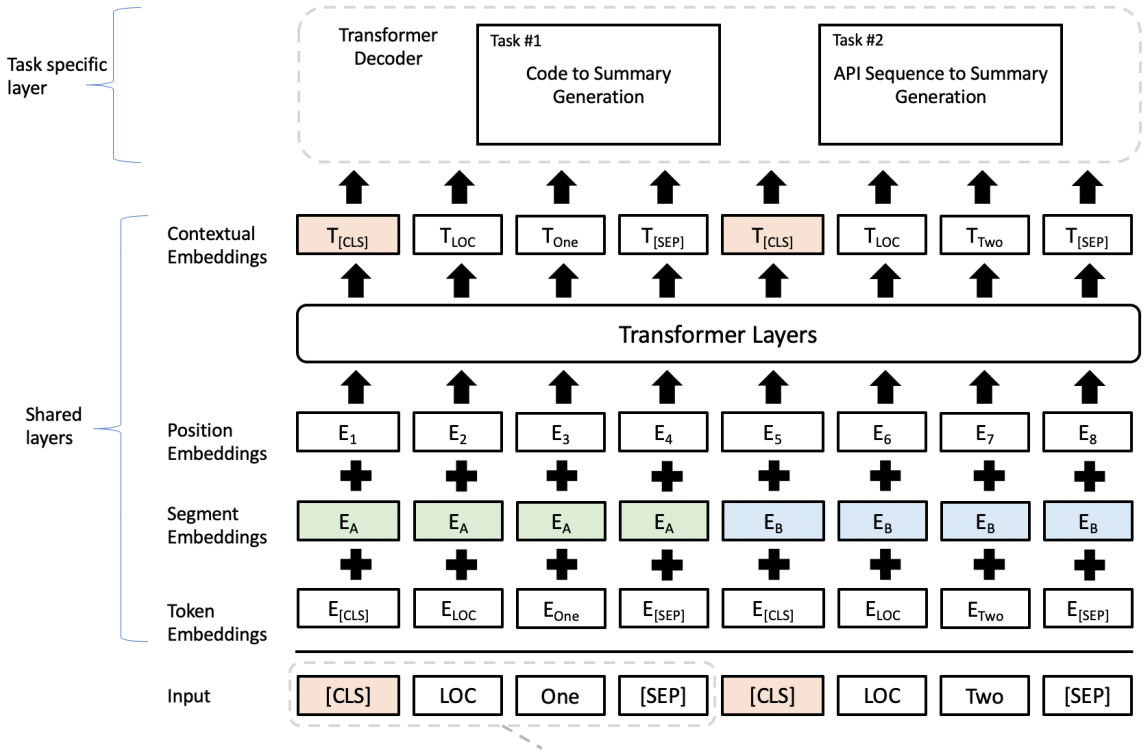

Figure 2: Overview of the proposed approach. Different from the original BERT structure, we prefix every line of code with a `[CLS]` symbol for learning the line of code representation. Each line of code also has a different type of segment embeddings. Our experimental results showed that the differentiation among different lines of code achieves a higher performance. Our decoder consists of a standard 6-layer Transformer decoder. On multi-tasking learning, our proposed model consists of training two different but similar tasks: 1) Code to Summary Generation, and 2) API sequence to Summary Generation.

trained in step s, and the other model is trained in step $s+1$. The result of the model in step s is then used for the model in step $s+1$. This cycle keeps on repeating until convergence. Our work does not require cycle dependency between two models but simultaneously trains two tasks, which is more efficient.

CodeBERT (Feng et al., 2020) and PYMT5 (Clement et al., 2020) present pre-trained models by using a multi-layer bidirectional transformer encoder (Feng et al., 2020) and a text-to-text transfer transformer T5 (Clement et al., 2020), respectively. CodeBERT (Feng et al., 2020) and PYMT5 (Clement et al., 2020) both support multiple downstream tasks such as code search (Feng et al., 2020), code generation (Clement et al., 2020), and code summarization (Feng et al., 2020; Clement et al., 2020). Different from these techniques, our model is designed for code summarization tasks specifically, thus provides better performance as our results show in Section 5[1]. Furthermore, our proposed approach can be combined with any pre-trained models, other than BERT, potentially improving upon their original performances.

3 Proposed Approach

As shown in Figure 2, our proposed model has a common encoder-decoder architecture, consisting of training two different but similar tasks: 1) Code to Summary Generation, and 2) API sequence to Summary Generation. Although we trained two different tasks, our main task is Code to Summary Generation. We first describe the encoder-decoder architecture, followed by the multi-task learning framework of the two tasks.

[1]For PYMT5, its data, model, and code are not publicly available. Furthermore, to re-train PYMT5, it is computationally very expensive to train as indicated in the PTMT5 paper i.e., it requires 16 V100 GPU with 32GB VRAM trained for 3 weeks. Thus, we were unable to re-train PYMT5 on our end.

3.1 Encoder Architecture

The large-scale pre-trained language models have shown remarkable performance in recent NLP studies. However, the use of such models is rarely studied in programming languages. In this work, we explore the potential of a popular pre-trained language model, BERT (Devlin et al., 2019). Specifically, we make use of the uncased base model. We have previously conducted our experiments using cased pre-trained models, taking into consideration camel cases. However, the results are not better than using uncased pre-trained models. Thus, we have omitted them. We are aware of a recent code representation model, CodeBERT (Feng et al., 2020). However, there has been no study on how a popular BERT-like structure that performs effectively in downstream NLP tasks can be leveraged in code summarization, given the language similarity between programming languages and the English language. In this study, we proposed the learning of line of code representation in a BERT-like encoder and showed that it achieves better performance than a vanilla BERT encoder.

In the original BERT model, every sentence is prefixed with the `[CLS]` token and ends with a `[SEP]` token. We observed that the indentation of code lines can have a special meaning for certain programming languages. For example, combining two different lines of code in Python may cause errors. Thus, we believed that a model may be able to train better if it can identify the line difference. In the encoder, our work models every line of code distinctly by inserting an additional `[CLS]` token at the start of every line of code. Similar to vanilla BERT, `[SEP]` is appended as the last token for every line of code. We use the original code indentation and we do not further process the code to conform to a certain indentation. In our early experiment, we have attempted to model different forms of whitespace indentation. However, these modelings do not improve the overall model performance. Thus, we exclude them in our final model design. Additionally, all the code and summary are lower-cased, and every non-alphanumerical symbol in the code is treated as a separate token.

The input $\boldsymbol{x} = x_1^j, x_2^j, ..., x_1^{j+1}, x_2^{j+1}, ..., x_m^{j+n-1}$ is a running sequence of code tokens that are arranged in lines of code (LOC) beginning from the top of every method/function. x_i^j denotes the i^{th} token of the j^{th} line of code. For example, Figure 3 shows a Java method consisting of three lines of code. The input will then be processed as

```
[CLS] public string printmystring ( )
{ [SEP] [CLS] return " hello world " ;
[SEP] [CLS] } [SEP]
```

where `public` and `string` refer to the second and the third token of the input (e.g., x_2^1 and x_3^1 in input $\boldsymbol{x}$ above). Note that the first token of every line of code (e.g., x_1^1) is `[CLS]`.

```
public String printMyString () {
    return "Hello World";
}
```

Figure 3: Example of a Java method

As shown in Figure 2, each token x_i for line j is assigned three kinds of embeddings: token embeddings, segmentation embeddings, and position embeddings. Token embeddings refer to the semantics of each token. Segmentation embeddings are used to distinguish between different lines of code. For example, for each LOC, the approach assigns segment embeddings E_A or E_B depending on whether the line of code is even or odd, as shown in Figure 2. Position embeddings indicate the position of each token within the line of code.

These three embeddings are added to a single input vector and fed into a bidirectional Transformer with multiple layers, i.e.,

$$\tilde{h}^l = LN(h^{l-1} + MHAtt(h^{l-1})) \quad (1)$$

$$h^l = LN(\tilde{h}^l + FFN(\tilde{h}^l)), \quad (2)$$

where $h^0 = x$ is the input vectors. The superscript l indicates the depth of the stacked layer. LN is the layer normalization operation (Ba et al., 2016) and $MHAtt$ is the multi-head attention operation (Vaswani et al., 2017). FFN is a Feed-Forward Network. As a result, the encoder generates an output vector T_i (shown in Figure 2) for each token with rich contextual information.

3.2 Decoder Architecture

Our decoder is a six-layered Transformer (Vaswani et al., 2017) initialized randomly. While our encoder is a pre-trained model, our decoder must be trained from scratch. This makes fine-tuning of BERT unsuitable. To mitigate this imbalance issue, Adam optimizer (Kingma and Ba, 2015) with different hyperparameter values β_1 = 0.9 and β_2 = 0.999 is used in the encoder and decoder, respectively.

Additionally, different warm-up steps and learning rates are imposed in the encoder and decoder, i.e.,

$$lr_E = \tilde{lr}_E \cdot min(step^{-0.5}, step \cdot warmup_E^{-1.5}) \quad (3)$$

$$lr_D = \tilde{lr}_D \cdot min(step^{-0.5}, step \cdot warmup_D^{-1.5}), \quad (4)$$

where lr_E and lr_D denote the learning rates for the encoder and decoder, respectively, and $warmup_E$ and $warmup_D$ denote the warmup steps for the encoder and the decoder, respectively. lr_E and $warmup_E$ are initialized to $2e^{-3}$ and 20,000, respectively, and lr_D and $warmup_D$ are initialized to 0.1 and 10,000 respectively. lr_E and $warmup_E$ are set lower than its decoder counterparts so that the encoder can be trained with more accurate gradients when the decoder is becoming stable (Liu and Lapata, 2019). For every task, we set the same learning rates and the warmup steps for both an encoder and a decoder.

3.3 Multi-task Learning

Our multi-task learning approach is similar to one designed for natural language (Liu et al., 2019). The lower layers in Figure 2 indicate the shared layers across all tasks, and the top layer represents task-specific outputs. The shared layers contain final contextual embeddings, which are the output of multiple stacked transformer layers. The input to the transformer layers is the summation of the token embeddings, segment embeddings, and position embeddings. The task-specific layer uses a transformer decoder for two different tasks where the input to the decoder is the contextual embeddings from the shared layers.

Algorithm 1 illustrates our multi-task learning procedure. During the multi-task learning, for every mini-batch in Task #1 and Task #2, the model is updated according to the objective of Task #1 and #2, respectively. Such setup has been reported to be effective and approximately optimize the sum of all multi-task objectives (Liu et al., 2019). We describe Task #1 and Task #2 in detail as follows.

Task #1: Code to Summary Generation

This task takes the code as the input and gives the summary as the target output. Each line of code is further prefixed with the `[CLS]` token for learning the line of code representation. The last token for every line of code is `[SEP]`. We use the cross-entropy loss as the objective function, i.e.,

$$\frac{1}{N}\sum_{i=1}^{N} y_i^{T1} log(\hat{y}_i^{T1}), \quad (5)$$

where y_i^{T1} denotes the target token of the summary at time step i for Task #1, and $\hat{y}_i^{T1}$ denotes the probability of generating the token for Task #1 at time step i. N is the total number of words generated.

Task #2: API sequence to Summary Generation

Task #2 is similar to Task #1 except that instead of taking every code token as input, API sequence is used as input. Furthermore, our approach does not distinguish between different lines of code in Task #2, and the entire API sequence of a function is treated as a single line of code. The objective function is also set as the cross-entropy loss, i.e.,

$$\frac{1}{N}\sum_{i=1}^{N} y_i^{T2} log(\hat{y}_i^{T2}) \quad (6)$$

where y_i^{T2} denotes the target token of the summary at time step i for Task #2, and $\hat{y}_i^{T2}$ denotes the probability of generating the token for Task #2 at time step i. N is the total number of words generated.

Algorithm 1 TRAINING AN MT MODEL

```
for all mini-batch do
    1. Compute Loss: L(Θ)
        L(Θ) = Eq. 5 for Task #1
        L(Θ) = Eq. 6 for Task #2
    2. Compute gradient: ∇ (Θ)
    3. Update model: Θ = Θ - ∇ (Θ)
end for
```

4 Experimental Setup

This section describes the datasets used in our experiments (Section 4.1), the different metrics used in the automatic evaluation (Section 4.2), and the qualitative evaluation (Section 4.3). The different baselines are discussed in Section 4.4 and the hyperparameters to our models are listed in Section 4.5.

4.1 Datasets

We made use of two common datasets, Java (Hu et al., 2018b) and Python (Miceli Barone and Sennrich, 2017; Wan et al., 2018). They have been widely used in previous work (Hu et al., 2018b; Wan et al., 2018; Wei et al., 2019; Ahmad et al., 2020). Each dataset consists of pairs of code and a

single sentence summary describing the code. All the datasets were split distinctly into Train, Validation and Test set (shown in Table 1). We use the exact same datasets in each split as previous studies (Wei et al., 2019; Ahmad et al., 2020) without any alteration.

Java The Java methods and their summary were collected from Java projects in Github from 2015 to 2016. The first sentence of every method in the Javadoc was extracted to be the ground truth code summary. As a result, each Java method forms a <code, summary> pair. Following prior work (Gu et al., 2016), the API sequence of a Java method was collected by parsing the Java method using Eclipse's JDT compiler[2], constructing an AST tree, and extracting the API sequence represented in the AST tree. The second column of Table 1 presents the data statistics of Java.

Python The python functions and their summary were collected from Python projects in Github in 2016. If a python function consists of a docstring, the first sentence of the docstring is treated as the ground truth code summary, and its corresponding function forms the code of the summary. Similar to Java, the Python code is first parsed into an AST tree using asttokens[3], which is a common library for transforming python code into the AST form. The API sequence of a Python function are then extracted from the AST tree. The third column of Table 1 shows the data statistics of Python.

Dataset	Java	Python
Train	69,708	55,538
Validation	8,714	18,505
Test	8,714	18,502
Unique tokens in code	66,650	307,596
Unique tokens in summary	46,895	56,189
Avg. tokens in code	120.16	47.98
Avg. tokens in summary	17.73	9.48

Table 1: Statistics of Java and Python datasets.

4.2 Metrics for Quantitative Analysis

We evaluate our approach using three widely used metrics in code summarization, as follows.

BLEU (Papineni et al., 2002) quantifies the lexical similarity of the generated summary to the ground truth summary by counting the common n-grams.

METEOR (Banerjee and Lavie, 2005) measures the alignment between the generated and the ground truth summary by exact, stem, synonym, and paraphrase matches between words and phrases.

ROUGE-L (Lin, 2004) measures the longest common subsequence overlap between the generated and ground truth summary, and focuses on recall scores.

4.3 Qualitative Analysis

We randomly select 200 generated summaries along with their original code, 100 pairs each for Java and Python, following similarly to prior research (Liu and Lapata, 2019; Grusky et al., 2018). Amazon Mechanical Turk (MTurk) workers were hired to rate the quality of the generated summaries. The MTurkers rated the summary voluntarily, and for each rated summary, the MTurkers are given a compensation of one cent. We used four common criteria to evaluate the summarization quality (Liu and Lapata, 2019):

Informativeness How well does the summary capture the key points of the code?

Relevance Are the details provided in the summary consistent with details in the code?

Fluency Are the summaries well-written and grammatically correct?

Comprehension Can the summaries helps in understanding the code?

Three different workers were required to rate each summary between one and five, where one is the worst and five is the best. We also ask the MTurkers for their Java/Python coding experience and if they understand the generated summaries and code.

In addition to the MTurk surveys, we performed additional analysis on the same set of code-summary pairs as those reviewed by MTurkers. The purpose is to further unravel the quality of our generated summaries by investigating the difference between generated and ground-truth summaries.

4.4 Baseline Models

We compare our approach with the following eight baseline models as seen in Table 2.

CODE-NN (Iyer et al., 2016) uses token embeddings as source code embeddings, and the overall model architecture is based on LSTM. Additionally, it uses a global attention mechanism that computes a weighted sum of the source code embeddings during the decoding process.

Tree2Seq (Eriguchi et al., 2016) uses a tree-like Sequence-to-Sequence model where the source

[2]https://www.eclipse.org/jdt/

[3]https://pypi.org/project/asttokens/

Model	Java			Python		
	BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
CODE-NN (Iyer et al., 2016)	27.60	12.61	41.10	17.36	9.29	37.81
Tree2Seq (Eriguchi et al. 2016)	37.88	22.55	51.50	20.07	8.96	35.64
RL+Hybrid2Seq (Wan et al., 2018)	38.22	22.75	51.91	19.28	9.75	39.34
DeepCom (Hu et al., 2018a)	39.75	23.06	52.67	20.78	9.98	37.35
API+CODE (Hu et al., 2018b)	41.31	23.73	52.25	15.36	8.57	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[1] (Feng et al., 2020)	12.83	10.12	25.26	16.25	13.52	27.66
CodeBERT[2] (Feng et al., 2020)	30.92	20.46	43.45	25.55	19.88	40.95
Ours w/o multi-task w/o LOC modeling	45.37	28.76	56.11	34.60	21.38	49.83
Ours w/o multi-task	46.27	28.40	56.68	35.01	22.34	50.10
Ours	**47.15**	**30.38**	**57.21**	**35.48**	**22.64**	**50.88**

Table 2: Comparison of our proposed approach with the baseline results. Our proposed model and ablation settings are shown in the bottom three rows. Our proposed model and ablation settings consistently achieve the state-of-the-art performance when compared with all the other baseline models in both the Java and Python datasets.

code is transformed into a structural tree representation, which is used as the input to the encoder.

RL+Hybrid2Seq (Wan et al., 2018) uses both code and the Abstract Syntax Tree of the corresponding code as the input to a Reinforcement Learning model.

DeepCom (Hu et al., 2018a) also uses both code and the Abstract Syntax Tree as input, but for a general Sequence-to-Sequence model. For the attention mechanism, it considers both code and Abstract Syntax Tree.

API+CODE (Hu et al., 2018b) uses API sequence and code summary to first train a Sequence-to-Sequence model. A secondary model is then created for the purpose of code to summary. In the secondary model, if the code is an API, its embeddings would be borrowed from the first model.

Dual Model (Wei et al., 2019) trains two models where the first model generates summary from code and the second model generates code from summary. The training is a cyclic process where the intermediate output of each trained model is used as the input to the other model.

Transformer (Ahmad et al., 2020) uses a transformer architecture for encoding code and decoding summary. In addition to encoding the absolute position of each code token, it leverages a pairwise relationship among all the code tokens via attention. The combination of the positional encoding and the pairwise relationship encoding forms a richer contextual positional vector.

CodeBERT (Feng et al., 2020) is a transformer-based neural architecture model for representing Programming Language (PL) and Natural Language (NL). It can perform multiple different types of downstream PL-NL tasks including code summarization. We compare our results with two CodeBERT variants, CodeBERT[1] and CodeBERT[2]. CodeBERT[1] tunes CodeBERT's Code Summarization task with CodeBERT's own training and validation datasets while CodeBERT[2] tunes with the common training and validation datasets presented in Table 1. For both, the default tuning settings recommended by the authors of CodeBERT are used. In testing, we use the same common test datasets in Table 1 for all models.

4.5 Hyperparameters

We applied dropout of probability 0.1 before all linear layers and used label smoothing (Szegedy et al., 2016) with smooth factor 0.1. During decoding, the beam size is set to 5. We follow prior work (Wei et al., 2019; Ahmad et al., 2020) to set the maximum length of code and summary to be 150 and 50, respectively. Other hyperparameters, including the number of epochs, are tuned based on the model performance on the validation set. All experiments are conducted on eight NVIDIA RTX 2080 GPUs during the four-week-long period.

5 Results

We provide the automatic evaluation of the baseline models and our proposed model in Section 5.1 followed by the human evaluation under Section 5.2. In addition to the human evaluation, we also provide several examples for additional qualitative analysis.

Java

```
/**
 * Generated Summary:
 *  creates and adds a new layout panel
 * Ground Truth Summary:
 *  lays out the panel
 */
private void initializeLayout(){
  GridLayout gl = new GridLayout(0,2);
  gl.setVgap(5);
  setLayout(gl);
  add(new JLabel("Frozen:"));
  add(frozenDD);
  add(new JLabel("UpperBound:"));
  add(tfUpBound);
  add(new JLabel("LowerBound:"));
  add(tfLowBound);
  add(new JLabel("Increment:"));
  add(tfIncrement);
  add(new JLabel("Delay:"));
  add(tfDelay);
  setBorder(
   BorderFactory.createEmptyBorder(5,5,5,5));
}
```

Python

```
def ListAllKeys(store,
                callback,
                prefix=None, marker=None,
                batch_size=1000):
  """
  Generated Summary:
   list all keys
  Ground Truth Summary:
   list all keys
  """
  keys = []
  done = False
  while (not done):
    new_keys = (yield gen.Task(
      store.ListKeys,
      prefix=prefix,
      marker=marker,
      maxkeys=batch_size))
    keys.extend(new_keys)
    if len(new_keys) < batch_size:
      break
    marker = new_keys[(-1)]
  callback(keys)
```

Figure 4: Examples of generated summary by our proposed multi-task model. The generated summaries of our proposed model have same semantics as the ground truth (human-written) summaries.

5.1 Quantitative Results

Table 2 shows the results of the baseline models (row two to nine) and our proposed model with two ablation settings (the bottom three rows). The automatic scores for the Java dataset are shown in columns two to four and for the Python dataset, they are shown in columns five to seven. The table shows that our proposed model and two ablation settings have consistently achieved higher performance than all the baseline models.

To evaluate the effectiveness of our LOC modeling and multi-task model discussed in Sections 3.1 and 3.3, respectively, we performed the ablation study by running the experiments without these two components. The second bottom row shows the result of our proposed model without the multi-task component. This means that the model considers only the single task for code to summary (i.e., Task #1 in Figure 2) without the task of API sequence to summary (i.e., Task #2 in Figure 2). The third bottom row shows the result of our proposed model without both multi-task and LOC modeling components. The model without the LOC modeling means that it treats the code as a contiguous sequence of tokens. In short, the comparison result between the second and third bottom rows show the effectiveness of our LOC modeling. The comparison result between the second and last bottom rows shows the effectiveness of our multi-task model. In the table, we omit the results of training a single-task model on API sequence to summary because different code may have the same API sequence, thus the model would not be well-trained. We have elaborated this with an illustration using Figures 1a and 1b in Introduction.

Table 2 shows that by considering learning the LOC modeling, the majority of the metrics are improved. For example, in the Java dataset, two (BLEU and ROUGE-L) out of three metrics in "Ours w/o multi-task" show improvement over "Ours w/o multi-task w/o LOC modeling". In the Python dataset, all three metrics have improved. This suggests that our proposed LOC modeling is effective. Furthermore, our proposed multi-task approach (last row in Table 2) scores the best performance in both the Java and Python datasets, achieving the new state-of-the-art.

5.2 Qualitative Results

Figure 4 shows the qualitative examples of the generated summaries. The first column consists of an example of Java – the creation of `GridLayout`. Although the generated summary and ground truth summary differ largely in terms of the unigram, they have the same semantics. The second column consists of an example of Python code on listing keys. Both the generated summary and the ground truth summary are identical. The figure shows that our generated summaries achieve good quality by producing the same semantics to the ground truth.

Table 3 shows the survey results from Amazon MTurkers on the generated summaries given the

	Info.	Relevance	Fluency	Compre.
Java	3.71	3.81	3.67	4.12
Python	3.05	2.95	2.92	3.24

Table 3: Qualitative results from the MTurker studies. Generally, MTurkers find that the generated summaries for Java are informative, relevant, and fluent, and both the Java and Python generated summaries can help in understanding the code.

Java and Python code. The majority of the MTurkers have (86.3%) Java experience and (72.2%) Python experience between 1 to 5 years, in the Java survey and Python survey, respectively. Only a minority of the MTurkers do not understand the code and the generated summary: 19.4% for Java and 27.3% for Python. On average, MTurkers found that the generated summaries for Java are informative, relevant, and fluent. For Python, MTurkers found the generated summaries are less informative, less relevant, and less fluent than summaries for Java. We believe that the main reason for the lower performance of Python might be due to the flexibility of the Python programming language (as compared to the Java programming language), which is dynamically typed, and it allows developers to write multiple different variants of code for the same functionality. For example, in Python, instead of having multiple lines of code in a loop, the developer can combine them into a single line for list comprehension. For both Java and Python, MTurkers found the generated summaries can help them understand the code better.

For the authors' analysis, the majority of the generated summaries produce the same meaning as the ground truth: 35% (Java) and 16% (Python) provide identical summaries to the ground truth, and 29% (Java) and 38% (Python) hold different structures but have the same semantics as the ground truth. Those yielding different meanings still achieve high quality generated summaries by missing just a few points from the ground truth (e.g., "delete and create a directory in ground truth" becomes "create a directory" in generated summaries) and by using slightly different adjective phrases that do not damage the key points (e.g., "latex preamble" in ground truth becomes "current preamble").

6 Conclusions

In this work, we proposed a novel and effective multi-task approach for generating summaries from code. Two different but similar tasks, 1) generating summaries from code, and 2) generating summaries from API sequence, are trained simultaneously. Our proposed model also considers modeling every line of code (LOC) whereas existing work treats code as a single contiguous sequence. To the best of our knowledge, this is the first work that utilizes a natural language-based pre-trained language model for a code summarization task. Our experimental results on two popular datasets, Java and Python, show that our proposed model performs better than all baselines, achieving the new state-of-the-art performances. Additionally, both the multi-task component and LOC modeling component of our proposed model are demonstrated to be effective. Furthermore, our proposed approach can be combined with any pre-trained models, other than BERT, potentially improving upon their original performances.

Acknowledgments. Mijung Kim is the corresponding author. This work was partially supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00075, Artificial Intelligence Graduate School Program (KAIST) and No. 2020-0-00368, A Neural-Symbolic Model for Knowledge Acquisition and Inference Techniques).

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv: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.

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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*, page 823–833.

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.

Max Grusky, Mor Naaman, and Yoav Artzi. 2018. Newsroom: A dataset of 1.3 million summaries with diverse extractive strategies. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 708–719.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE.

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

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-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.

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.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network.

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

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.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4487–4496, Florence, Italy. Association for Computational Linguistics.

Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language

summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE.

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.

Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

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.

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*, pages 6563–6573.

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