Pointing to Subwords for Generating Function Names in Source Code

Shogo Fujita[1], **Hidetaka Kamigaito**[1], **Hiroya Takamura**[1,2] **and Manabu Okumura**[1]
[1]Tokyo Institute of Technology
[2]National Institute of Advanced Industrial Science and Technology (AIST)
{fujisyo,kamigaito,oku}@lr.pi.titech.ac.jp
takamura@pi.titech.ac.jp

Abstract

We tackle the task of automatically generating a function name from source code. Existing generators face difficulties in generating low-frequency or out-of-vocabulary subwords. In this paper, we propose two strategies for copying low-frequency or out-of-vocabulary subwords in inputs. Our best performing model showed an improvement over the conventional method in terms of our modified F1 and accuracy on the Java-small and Java-large datasets.

1 Introduction

Programmers often share source code on sharing services such as GitHub.[1] Since they can freely define function names in the source code, the names are not necessarily reminiscent of the actual behavior of the functions. For example, the function in Figure 1 returns the index of `elem` whose `elem.key` is the same as `target_key`. However, the function name would be inappropriate as it implies that the function returns the value of the object. Such a function name adversely affects readability and sometimes causes bugs, especially in collaborative environments. A proper function name such as `indexOfTarget` in this case, instead of `getTargetValue`, can help programmers understand the code efficiently and avoid possible bugs (Takang et al., 1996; Binkley et al., 2013). Automatically generating such function names has been studied as a generation task in natural language processing (Iyer et al., 2016).

```
int getTargetValue(int target_key) {
  int index=0;
  for (Object elem: this.elements) {
    if (elem.key == target_key) {
      return index;
    }
    index++;
 }
 return -1;
}
```

Figure 1: Example of a function and its inappropriate name. (Java)

Recently, various neural network-based approaches have been proposed to solve this problem by generating a function name from given source code (Allamanis et al., 2016; Alon et al., 2018; Fernandes et al., 2018). In these approaches, a function name is treated as a sequence of subwords (`get`, `Target` and `Value` in Figure 1). Since these approaches heavily rely on a subword-based predefined dictionary to generate a function name, it is difficult to generate a function name containing low-frequency or unknown subwords.

To solve this problem, we propose a method for outputting low-frequency or unknown words using a copy mechanism corresponding to a tree structure, and a method for replacing a specific word with a special token. We extend code2seq (Alon et al., 2019a) by using these methods. Code2seq converts source code into an tree-structured representation, called Abstract Syntax Tree (AST), before encoding.

[1]https://github.com

Proceedings of the 28th International Conference on Computational Linguistics, pages 316–327
Barcelona, Spain (Online), December 8-13, 2020

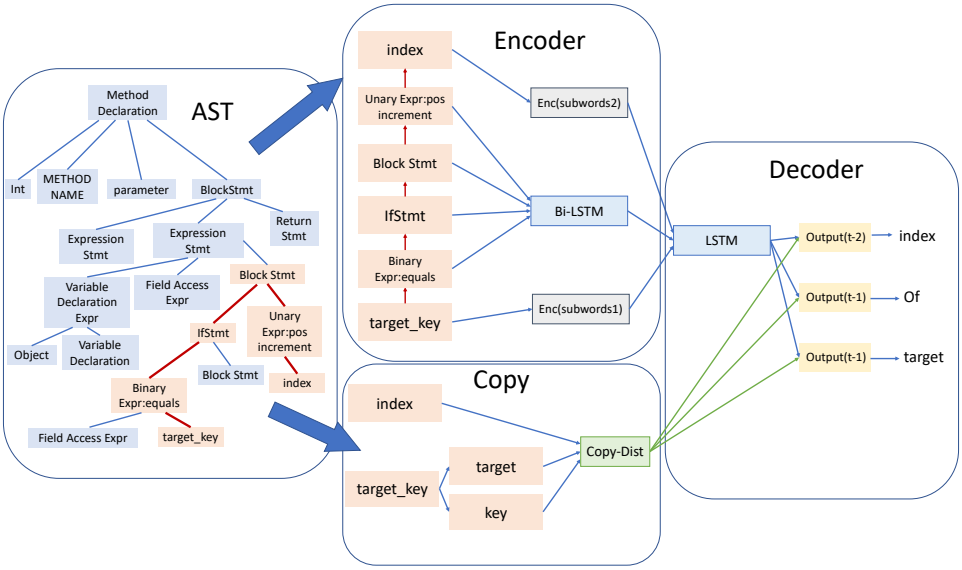

Figure 2: Overview of the function naming with our model.

The input for the encoder is not just a sequence of tokens but a set of paths from a leaf to another leaf in the tree. Thus, the existing copy mechanisms (Gulcehre et al., 2016; Gu et al., 2016; Yang et al., 2018; Hsu et al., 2018; Cohan et al., 2018) cannot be directly applied.

We observed that our best-performing model was the one that uses a combination of a hierarchical copy mechanism and a strategy to replace the most frequent word in an input snippet of source code with a delexicalized placeholder. In particular, the score of the best-performing model was increased in terms of our modified F1 and accuracy, calculated on the Java-small and Java-large[2] datasets by Alon et al. (2019a).

2 Code2seq

We first describe code2seq (Alon et al., 2019a), an existing model that we extend in this paper. Code2seq first converts an input snippet of source code into an AST, a tree-structured data representation given by a parser in a compiler. After that, an encoder-decoder model is used to generate a function name from the AST. We describe ASTs and the architecture of the base model below.

2.1 Abstract Syntax Tree (AST)

An AST is an intermediate representation used when source code is analyzed by a compiler. The left part of Figure 2 shows an AST obtained from part of the source code in Figure 1. The leaves in the tree correspond to the strings that appear in the source code, and the non-terminal nodes are defined by the compiler. The AST is obtained by using JavaParser.[3]

2.2 Encoder

As shown in Figure 2, code2seq takes an obtained AST as an input. It then extracts all possible shortest paths from a leaf to another leaf from the AST. Each path can be considered to be a sequence of nodes in the AST. Next, it vectorizes two leaves and a sequence of non-terminal nodes on the shortest path. Each leaf v is split into subwords and is converted into $\beta(v)$, the vector of the leaf that is defined as the sum of e_w^{sub}, the embedding vectors of the subwords w.

The sequence of non-terminal nodes is converted into a vector by using a bidirectional long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) based encoder: $\overrightarrow{h_t} = \texttt{LSTM}(\overrightarrow{h}_{t-1}, x_t)$

[2]https://github.com/tech-Srl/code2seq#datasets
[3]https://github.com/javaparser/javaparser

and $\overleftarrow{h_t} = \texttt{LSTM}(\overleftarrow{h}_{t+1}, x_t)$, where x_t is the embedding of a non-terminal node. The encoded vectors are concatenated as: $\gamma(v_1 \cdots v_n) = [\overrightarrow{h}_n; \overleftarrow{h}_1]$, where ';' represents the concatenation of two vectors. The vectors representing the two leaves and the vector representing the sequence of internal nodes are combined as follows:

$$q_{v_0,v_{n+1}} = tanh(W_{in}[\gamma(v_1 \cdots v_n); \beta(v_0); \beta(v_{n+1})]),$$

where W_{in} is a matrix of a linear transformation of the concatenated vectors, and v_0 and v_{n+1} are the leaves of the beginning and end of the sequence, respectively.

2.3 Decoder with Attention

The decoder inherits the averaged vector of all possible paths between the leaf nodes in the AST as an initial state s_0. To prevent the computational space from becoming too large, the maximum number of paths is set to 200; if there are more than 200 paths, 200 paths are randomly selected. At each time step t, the decoder calculates the current hidden state $s_t = LSTM(s_{t-1}, y_{t-1})$, where y_{t-1} is the embedding of the predicted subword in the previous time step. By using s_t, the decoder calculates the attention weights (Luong et al., 2015) on the paths, each of which connects two leaf nodes. The weight on the r-th path is defined as follows:

$$a_r^t = \frac{\exp(d_a^T \tanh(W_a[s_t; q_r]))}{\Sigma_{r'} \exp(d_a^T \tanh(W_a[s_t; q_{r'}]))}, \tag{1}$$

$$p_{voc}(w) = \sum_{i:w=w_i} \delta_i^T softmax(W_l[\Sigma_r a_r^t q_r; s_t]), \tag{2}$$

where q_r is the vector representation of the r-th path in the encoder, W_a is a weight matrix for the linear transformation, and d_a is a parameter vector. Finally, the output layer calculates the label distribution at time step t as $p_{voc}(w)$, where W_l is a weight matrix. δ_i is a one-hot vector, where only the i-th element is 1, and the others are 0. w_i is the i-th subword in the vocabulary.

3 Proposed Method

We extend code2seq by adding a mechanism that generates low-frequency or out-of-vocabulary subwords. (Figure 2) We propose two methods: the first one replaces the most frequent subword with a delexicalized placeholder; and the second one is a hierarchical copy mechanism.

3.1 Placeholder for Most Frequent Subword

In this method, we replace the most frequent subword in each input snippet of the source code with a placeholder `MFS` in the training data. This idea is based on the observation that 31.26% of function names in the training data include the most frequent subwords in the input snippets. The existing methods have naively replaced all out-of-vocabulary subwords with a special tag, `UNK`. We argue that such a strategy causes a lack of information for the tokens that should be included in the function name. In comparison, our model knows which subwords are important even if they are out-of-vocabulary subwords, regarding the most frequent one as important. Thus, the model can more properly identify the important parts in the source code by assuming that a sequence containing the placeholder `MFS` is important. When `MFS` appears in the output in the generation phase, we replace it with the original subword.

3.2 Hierarchical Copy Mechanism

In this section, we describe our proposed method with a hierarchical copy mechanism (Figure 3). We observe that 71.42% of function names in the training data include at least one of the subwords in the input source code. This led us to the idea of integrating a copy mechanism into code2seq. The final probability of generating subword w is the weighted sum of the probability p_{voc} of generating the subword from the vocabulary and the probability p_{copy} of copying the subword from the input:

$$p(w) = p_{gen} p_{voc}(w) + (1 - p_{gen}) p_{copy}(w), \tag{3}$$

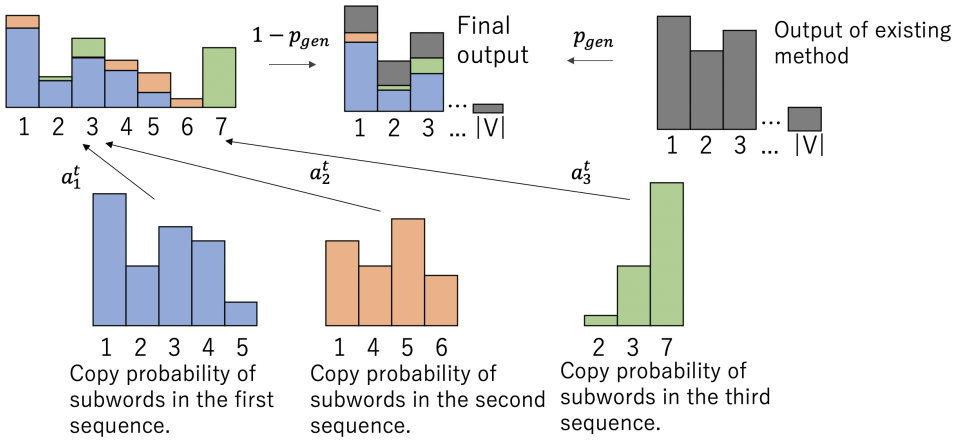

Figure 3: Hierarchical copy mechanism. $|V|$ represents the vocabulary size.

where p_{gen} is the probability of selecting p_{voc}.

Conventional copy mechanisms cannot be directly applied to our task because their input is assumed to be a sequence of words, while the input in our setting is a tree or a set of paths containing tokens. In particular, we propose a copy mechanism for copying subwords at leaf nodes. In our model, to calculate the copying probability $p_{copy}(w)$, our hierarchical method combines two weights, the weight a_r^t on the r-th path and the weight $b_{r,j}^t$ on the j-th subword on the r-th path, $w_{r,j}$:

$$p_{copy}(w) = \sum_r \sum_{j:w=w_{r,j}} a_r^t b_{r,j}^t. \tag{4}$$

We use the conventional attention weights (Luong et al., 2015) for a_i^t, as described in Section 2.3; and $b_{i,j}^t$ is calculated as follows:

$$b_{r,j}^t = \sum_j \delta_j^T softmax(h_{ctx}^T E_r^{sub}), \tag{5}$$

$$h_{ctx} = W_h a^t + W_s s_t + W_x e_{y_{t-1}}^{sub} + W_c g_t, \tag{6}$$

$$g_t = \sum_{k=0}^{t-1} a^k. \tag{7}$$

In the equation above, E_r^{sub} is a matrix consisting of the embeddings of all subwords on the r-th path. δ_j is a one-hot vector, where only the j-th element is 1, and the others are 0. a^t is a vector whose elements are the attention weights a_r^t for each path r at time step t, and $e_{y_{t-1}}^{sub}$ is an embedding of the previous output subword. Thus, g_t is a vector storing the sum of all attention weights at the previous time steps for every path. W_h, W_s, W_x, W_c are weight matrices, and w'_h, w'_s, w'_x, w'_c are weight vectors for the linear transformation.

p_{gen} is then calculated as follows:

$$p_{gen} = sigmoid(h'_{ctx}), \tag{8}$$

$$h'_{ctx} = {w'_h}^T a^t + {w'_s}^T s_t + {w'_x}^T e_{y_{t-1}}^{sub} + {w'_c}^T g_t. \tag{9}$$

4 Experiments

4.1 Experimental Settings

We evaluated our approaches on the following two datasets: Java-small and Java-large.[4] Java-small consists of 691,974 functions for training, 23,844 for development, and 57,088 for testing. Java-large

[4]https://github.com/tech-Srl/code2seq#datasets

consists of 15,344,512 functions for training, 320,866 for development, and 417,003 for testing. The models for comparison are as follows:

- **Code2seq**
  We described the model in Section 2. We reran the code[5] of Alon et al. (2019a).

- **Copy**
  This is a 2-layer LSTM-based pointer-generator model (See et al., 2017). We experimented with OpenNMT-py[6] with the copy_attn option.

- **Pointer**
  This is a variant of our hierarchical copy mechanism. Following the decoder of Fernandes et al. (2018),[7] this model only points to tokens (Vinyals et al., 2015) and does not generate any tokens. This model was prepared to verify the report of Fernandes et al. (2018) that a pointer-network works effectively and yields higher F1 scores than code2seq on the function naming task.

For training all the models, we used momentum-SGD (Qian, 1999) as an optimizer. The batch size was set to 256, and the dimension of subword embeddings was 128. The dimension of the hidden layer in the encoder was set to 128, and that in the decoder was set to 320. As a preprocessing, We split function and variable names in the source code into a sequence of subwords at the positons just before an uppercase character follows lowercase characters because programmers generally use camel case when writing code with Java. Long variable names were truncated to have at most 6 subwords. We used only the paths that had less than 9 subwords. We used TensorFlow to implement our models.

We used F1 as an evaluation metric, following Alon et al. (2019a), and added accuracy as another.

Furthermore, to correctly evaluate outputs with repeating tokens, we also used modified-F1 (F1**), calculated with the *modified unigram precision* of Papineni et al. (2002) and *unigram recall* of Lin (2004). F1** can prevent the models that repeatedly output subwords in the Gold function name from unreasonably obtaining high scores.

We calculated the above metrics on the basis of the number of subwords. The accuracy measure was defined to be the number of correctly generated function names divided by the total number of test instances. Here, we supposed an output is correct only if it is completely the same as the gold function name, while we calculated the other metrics by counting the overlap of subwords between generated function names and gold function names. We trained and evaluated each model three times and computed the averaged scores.

4.2 Results

Table 1 shows the results. On Java-small, our models scored higher than code2seq in all the metrics. In particular, F1** increased by 2.21 points with the replacement strategy, 4.86 points with the model with our copy mechanism, and 4.65 points with the combination. However, our best model had a lower F1 score than Pointer. This is consistent with the report of Fernandes et al. (2018), indicating that function names commonly contain many subwords included in the given source code. The reason why F1** for Pointer was significantly lower than F1 is probably that it repeatedly outputs the same tokens. The increase of the repetition may be caused by copying subwords from the small vocabulary included in the inputs. These results suggest that the models for this task need to generate subwords not included in the given source code in order to correctly generate function names. Moreover, the decoder part of seq2seq models is essentially the same as a unidirectional language model. For that reason, the vanilla decoder requires a large amount of training data for generating various tokens in the output. The pointer network can help the decoder to generate various tokens without training token embeddings in the decoder side. Thus, the pointer network can work even with a small amount of training data.

[5]https://github.com/tech-Srl/code2seq

[6]https://github.com/OpenNMT/OpenNMT-py

[7]They reported the state-of-the-art F1 scores on the Java small dataset. However, we could not reproduce their results with their code (https://github.com/CoderPat/structured-neural-summarization); other researchers also reported that they could not.(https://github.com/CoderPat/structured-neural-summarization/issues/25)

Corpus	Model	F1	F1**	Acc
Small	Code2seq∗	43.02	–	–
	Code2seq	42.81	40.69	16.20
	Copy	32.11	31.94	16.33
	Pointer	**47.91**	23.24	5.49
	Ours	47.52	45.34	**21.27**†
	w/o HierCopy	44.45	42.90	17.32
	w/o Replace	47.08	**45.55**†	19.13
Large	Code2seq∗	59.19	–	–
	Code2seq	59.00	58.16	35.75
	Copy	49.16	48.97	30.14
	Pointer	53.62	27.83	4.28
	Ours	58.96	**58.43**†	36.41
	w/o HierCopy	58.69	57.97	35.45
	w/o Replace	**59.59**†	58.40	**36.61**†

Table 1: Evaluation results. '∗' indicates the reported scores in the paper (Alon et al., 2019a). w/o indicates our model without the corresponding method. The highest score in each metric is shown in bold. † indicates that the difference from the best baseline was statistically significant with the paired bootstrap resampling method (Koehn, 2004) ($p < 0.001$).

Corpus	Target	Model	F1**	Acc
Small	Low	Code2seq	0.08	0.03
		w/o Replace	**18.47**†	**10.69**†
	Gen	Code2seq	27.74	22.46
		w/o Replace	**30.40**†	**24.20**†
Large	Low	Code2seq	20.56	11.49
		Ours	**31.21**†	**20.74**†
	Gen	Code2seq	40.19	**35.90**†
		Ours	**40.35**†	35.65

Table 2: Additional results for different types of target subwords. Low means only words that appear with a probability of less than 0.0001% in each corpus. About 3% of the function names contained one or more such words. Gen means only words that do not appear in input source code. Almost 8% of the function names contained one or more such words. Accuracy was calculated only with instances that contain at least one target subword.

In contrast, our models significantly outperformed Pointer in terms of F1** and accuracy. Our replacement strategy contributed little to F1** but significantly to accuracy. This is probably because our copy mechanism is effective in the decoding, whereas the replacement of the most frequent subwords helps to capture important information of the input in the encoder part. Thus, the combination can help both the encoder and decoder.

On Java-large, our models (the combination and with a copy mechanism) scored the highest in both F1** and accuracy among all the models. F1** increased by 0.27 points for the model with our copy mechanism, but the replacement strategy did not contribute at all. This shows that replacing the most frequent subwords with a special token leads to ignoring their original meaning, that causes a disadvantage in a large corpus. These results show the effectiveness of our hierarchical copy mechanism.

To check whether our best model can actually handle low-frequency or unknown subwords, we compared the best baseline and our models with the highest F1** score only on the subwords that appeared with a probability of less than 0.0001% (Low in Table 2). On Java-small, code2seq could hardly handle the words with the probability less than 0.0001%. In contrast, our method could output low-frequency words. On Java-large, while code2seq could output some infrequent words, our method handled infrequent and out-of-vocabulary words better.

To investigate the importance of generation rather than copying, we examined the performances of the best baseline and our models with the highest F1** score only for subwords not included in the input (Gen in Table 2). On Java-small, our proposed method outperformed code2seq even in cases where we focus only on subwords not included in the input. This is probably because the copy mechanism makes it easier to learn attentions with a small dataset. On Java-large, our method outperformed code2seq in F**, but did not outperform it in accuracy. It seems that our method emphasizes copying too much because function names tend to contain subwords in the input. The scores for Pointer were always almost 0 on both datasets in the Gen setting because it cannot generate any subwords, even though it achieved the

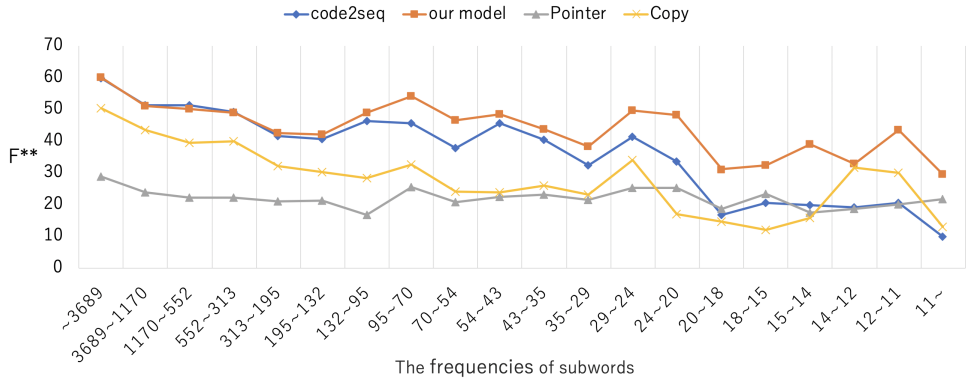

Figure 4: F** scores for subwords with various frequencies.

highest F1 score on Java-small. In contrast, comparing between Tables 1 and 2, the scores for our best model did not drop significantly even for the subwords not included in the input. These results suggest that the generation mechanism is necessary for the function naming task.

4.3 Analysis

4.3.1 F** Scores for Subwords with Different Frequencies

We analyzed the relationship between F** scores of each model and the frequencies of subwords. For this purpose, we first sorted subwords by their frequencies and after that, we split them into 20 classes equally. We then calculated F** scores for the subwords in each class separately.

Figure 4 shows the F** scores of each model for each class. The leftmost class is the most frequent subwords, and the rightmost is the least frequent subwords. The F** scores tend to decrease when the frequencies of subwords decrease. This observation supports our assumption that infrequent subwords are difficult to predict. Overall, the F** score for the leftmost class is almost the same as the result in Table 1 because the subwords in the class are the majority of the test data.

Code2seq and our model scored almost the same when frequencies of subwords are greater than 132. On the other hand, if the frequencies of subwords are less than 132, our model achieved higher F** scores than code2seq. Copy, which regards the source code as a sequence, scored lower than code2seq when the frequencies of subwords are greater than 14. If the frequencies of subwords are less than 14, Copy achieved higher scores than code2seq. These results indicate that even a vanilla copy mechanism can handle low-frequency subwords. However, our model achieved higher F** scores than Copy. These results indicate that our copy mechanism, which considers the abstract syntax tree, can handle low-frequency subwords better than the vanilla copy mechanism. Pointer, which does not have the generation mechanism in the decoder, scored lower than the other methods in high-frequency subwords. On the other hand, its F** scores for subwords whose frequencies are more than 11 and less than 20 were almost the same as the scores of code2seq, and the F** score of Pointer for subwords whose frequencies are less than 11 was significantly higher than the score of code2seq. Thus, copying subwords is more useful than generating subwords for infrequent subwords. Different from the other models, our model achieved the highest F** scores for subwords whose frequencies are less than 11. This result indicates that the substitution of the most frequent subword is also useful for infrequent subwords.

From these results, we can further conclude that our proposed hierarchical copy mechanism can handle low-frequency subwords in this task, compared with other baselines.

4.3.2 Outputs of each model

The top box of Figure 5 shows a function that checks whether binary data with a predefined name such as `busybox` and `toybox` exist in the root directory. Table 3 lists generated method names from each model. Because `busybox` is a low-frequency word, code2seq did not generate it. Copy generated `empty`, which does not appear in Gold. Pointer copied `busybox` but it outputted the same word repeatedly. In contrast, the output of our method is correct.

```
public static boolean METHOD_NAME(boolean includeToybox) {
    if(includeToybox) {
        return (findBinary("busybox", true)).size() > 0 ||
          (findBinary("toybox", true)).size() > 0;
    } else {
        return (findBinary("busybox", true)).size() > 0;
    }
}
```

is busybox available

```
boolean METHOD_NAME(SingleAnalysis result, String morphemeName) {
  for (MorphemeData forms : result.getMorphemeDataList()) {
    if (forms.morpheme.id.equalsIgnoreCase(morphemeName)) {
      return true;
    }
  }
  return false;
}
```

contains morpheme

Figure 5: Sample inputs.

The bottom box of Figure 5 is a function that checks whether the element of the first argument object has a morpheme of the second argument. Table 4 lists generated function names from each model. Code2seq wrongly generated `data` instead of `morpheme`. Copy generated `morpheme` correctly but the generated function name is not correct. Pointer successfully copied `morpheme` but it also copied wrong words that are not related to the given function. On the other hand, our method generated the function name which has a similar meaning to the one of Gold by replacing the most frequent subword in the given function. Specifically, our method replaced `MFS` with `morpheme` because `morpheme` is the most frequent subword in this function.

As illustrated in the output from code2seq in Table 4 that `morpheme` is generated as `data`, low-frequency words in a function might be replaced with more general-purpose words to explain how the function works. However, if those words are replaced with the general words, many function names would become the same, and it would be difficult to differentiate between them. Therefore, it is necessary to avoid using the general subwords such as `data` for generating function names. In that regard, our method is considered to be more practical because it can replace low-frequency words with MFS if they appear most frequently in the function.

Method	Output
Gold	is busybox available
Code2seq	is available
Copy	is empty
Pointer	is busybox available busybox available busybox
Our model	is busybox available

Table 3: Outputs for the top box of Figure 5

Method	Output
Gold	contains morpheme
Code2seq	has data
Copy	is morpheme
Pointer	single analysis morpheme morpheme morpheme name morpheme
Our model	has morpheme

Table 4: Outputs for the bottom box of Figure 5

5 Related Work

There have been a lot of research efforts on tasks where source code is the input. Hindle et al. (2012) and Babii et al. (2019) constructed language models for the source code. Raychev et al. (2015) proposed a method for outputting variable names in the source code.

Iyer et al. (2016) proposed a model to summarize the behavior of functions in the source code. Loyola et al. (2017) proposed a method for generating descriptions of source code changes.

While these studies focus on source code as an input, their outputs are not function names.

As a method for representing source code, Allamanis et al. (2015b) converted a snippet of the source code into AST and proposed a method for generating a short description of the behavior of the snippet. Allamanis et al. (2018) later proposed a method for detecting inappropriate variable names using AST. We also used AST to represent the input snippet of source code while many other researches treat the source code as a sequence of tokens.

Regarding studies on function name generation, Allamanis et al. (2015a) proposed a method for generating function names using a stochastic language model that takes a sequence of tokens as an input, while we used a set of paths in AST. Alon et al. (2018) formalized function name generation as a classification problem. Alon et al. (2019b) treated the same task as a sequence generation problem and proposed code2seq. In this paper, we proposed several extensions to code2seq.

Xu et al. (2019) used a hierarchical attention network for function name generation. In this model, the important information of the lower layer is passed to the upper layer by a recursive network. Our model also took into account the hierarchical structure in our copy mechanism, as described in Section 3.2.

We extended code2seq by adding the ability to copy subwords in the input source code. The copy mechanism is a technique that copies subwords in the input to the output (Gu et al., 2016; Gulcehre et al., 2016).

Copy mechanisms have been shown to be effective in many tasks such as question-answering (He et al., 2017), document summarization (See et al., 2017), headline generation (Nallapati et al., 2016) and question generation (Zhao et al., 2018). The existing copy mechanisms (Nallapati et al., 2016) presuppose a sequence of words as an input. Although Yang et al. (2018) and Hsu et al. (2018) proposed a copy mechanism with hierarchical attention networks at word and sentence levels and Cohan et al. (2018) proposed a copy mechanism with hierarchical attention networks at word and section levels, they both assumed the input is a sequence of words, sentences, or sections. Thus, their copy mechanisms cannot be directly applied to our setting because each input is assumed to be a set of paths in AST. Fernandes et al. (2018) proposed a method for a function naming task using copy mechanisms. They focused on extending the encoder, while we focused on extending the copy mechanism. Our method used a hierarchy of copy layers rather than a single copy layer.

6 Conclusion

This paper dealt with the function name generation task. We proposed two methods for including low-frequency or out-of-vocabulary subwords: replacing the most frequent subword in an input snippet of source code and with a hierarchical copy mechanism. Our models outperformed the existing methods in terms of our modified F1 and accuracy.

Our proposed copy mechanism is applicable to tree-structured inputs such as discourse structures, cooking recipes, and social network services. Moreover, replacing the most frequent subword seems to be useful in tasks where the vocabulary is relatively small.

There remain two major issues to address. The first is the need for better evaluation metrics. We believe that this task requires a metric that can accept synonyms such as METEOR (Banerjee and Lavie, 2005). However, some words that are considered synonymous in WordNet[8] are used differently in the context of source code. For example, `increment` is an operation that increases the value of a variable by 1 in source code. It cannot be replaced with a word such as `increase`, even if they are synonymous with each other. Therefore, we need an evaluation metric that takes into account the subtle difference between synonyms.

The second is to consider context in source code. Our approach generates function names only from the information inside the function. However, the behavior of other functions and the information on the objects handled by the function are important factors in generating the function name, because the function is called somewhere in the code. Therefore, automatic generation of function names can be made more practical by considering the context in the source code.

[8] https://wordnet.princeton.edu/

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015a. Suggesting accurate method and class names. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 38–49. ACM.

Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015b. Bimodal modelling of source code and natural language. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France. Proceedings of Machine Learning Research.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2091–2100, New York, New York, USA, 20–22 Jun. PMLR.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 404–419. Association for Computing Machinery.

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

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. Code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 40:1–40:29, New York, NY, USA, January. Association for Computing Machinery.

Hlib Babii, Andrea Janes, and Romain Robbes. 2019. Modeling vocabulary for big code machine learning. *Computing Research Repository*, abs/1904.01873.

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, June. Association for Computational Linguistics.

Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, Apr.

Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, New Orleans, Louisiana, June. Association for Computational Linguistics.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. In *Proceedings of International Conference on Learning Representations*.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany, August. ACL.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the unknown words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 140–149, Berlin, Germany, August. Association for Computational Linguistics.

Shizhu He, Cao Liu, Kang Liu, and Jun Zhao. 2017. Generating natural answers by incorporating copying and retrieving mechanisms in sequence-to-sequence learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 199–208, Vancouver, Canada, July. Association for Computational Linguistics.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA. IEEE Press.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November.

Wan-Ting Hsu, Chieh-Kai Lin, Ming-Ying Lee, Kerui Min, Jing Tang, and Min Sun. 2018. A unified model for extractive and abstractive summarization using inconsistency loss. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 132–141, Melbourne, Australia, July. Association for Computational Linguistics.

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

Philipp Koehn. 2004. Statistical significance tests for machine translation evaluation. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 388–395, Barcelona, Spain, July. ACL.

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

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292, Vancouver, Canada, July. Association for Computational Linguistics.

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

Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Çağlar Gulçehre, and Bing Xiang. 2016. Abstractive text summarization using sequence-to-sequence RNNs and beyond. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, Berlin, Germany, August. ACL.

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, July. Association for Computational Linguistics.

Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "big code". *SIGPLAN Notices*, 50(1):111–124, January.

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, July. ACL.

Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4:143–167.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2692–2700, Cambridge, MA, USA. MIT Press.

Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pages 10–21, New York, NY, USA. Association for Computing Machinery.

WeiJun Yang, ZhiCheng Tang, and XinHuai Tang. 2018. A hierarchical neural abstractive summarization with self-attention mechanism. In *2018 3rd International Conference on Automation, Mechanical Control and Computational Engineering (AMCCE 2018)*. Atlantis Press.

Yao Zhao, Xiaochuan Ni, Yuanyuan Ding, and Qifa Ke. 2018. Paragraph-level neural question generation with maxout pointer and gated self-attention networks. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3901–3910, Brussels, Belgium, October-November. Association for Computational Linguistics.