Code Generation from Supervised Code Embeddings

Han Hu[1]*, Qiuyuan Chen[2]*, and Zhaoyi Liu[3]*

[1] School of Software, Tsinghua University, Beijing, China
hh17@mails.tsinghua.edu.cn

[2] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
chenqiuyuan@zju.edu.cn

[3] School of Shenzhen Graduate, Peking University, Shenzhen, China, 518055
1701213615@sz.pku.edu.cn

Abstract. Code generation, which generates source code from natural language, is beneficial for constructing smarter **I**ntegrated **D**evelopment **E**nvironments (IDEs), retrieving code more effectively and so on. Traditional approaches are based on matching similar code snippets, and recently researchers pay more attention to machine learning, especially the encoder-decoder framework. Faced with code generation, most encoder-decoder frameworks suffer from two drawbacks: a) The length of the code snippet is always much longer than the length of its corresponding natural language, which makes it hard to align them, especially for encoders at word level; b) Code snippets with the same functionality could be implemented in various ways, even completely different at word level. For drawback a), we propose a new **S**upervised **C**ode **E**mbedding (SCE) model to promote the alignment between natural language and code. For drawback b), with the help of **A**bstract **S**yntax **T**ree (AST), we propose a new distributed representation of code snippets which overcomes this drawback. To evaluate our approaches, we build a variant of the encoder-decoder model to generates code with the help of pre-trained code embedding. We perform experiments on several open source datasets. The experiment results indicate that our approaches are effective and outperform the state-of-the-art.

Keywords: Code Generation · Code Embedding · Supervised Learning

1 Introduction

Generating code through natural languages (NL) is considered to be an important future direction of programming. On the one hand, it can lower the threshold of programming and facilitate programming process. On the other hand, it makes programmers more productive, for example, by generating non-core code automatically, which allows programmers to focus more on the core code. So using NL to map complex operations to basic code blocks receives tremendous interest and has shown great benefits.

Traditional code generation approaches are usually based on matching similar code snippets [7,6,5]. Recently, many researchers pay more attention to generating code by machine learning. Some researchers try to bridge the gap between two corpora by utilizing rich, existing code bases and program contexts [8,16]. Some researchers utilize a

* equal contribution

standard or a variant encoder-decoder model to map NL to a snippet of executable code directly [17,9,4,1,10,8].

Most existing approaches regard a code snippet as a simple plain sequence of words, without taking features of itself into account. In this way, current approaches suffer from two drawbacks: a) the length of code snippets usually differ a lot from that of NL, it seems a tough work for regular encoder-decoder models or attention mechanism to align them. b) Code snippets usually contain multiple functional processes, and the same functionality could be implemented in various ways. Mapping directly is likely to be disordered because there is no strict bijection between two corpora (i.e., code snippets and the corresponding natural language).

Faced with the drawback a), we propose a **S**upervised **C**ode **E**mbedding (SCE) model to pre-train distributions of code and NL at the same time, which helps align NL and code better. Faced with b), only considering the word-level feature of the code is far from enough, for example, Figure 1(a) and Figure 1(b) show an example of two Java functions which have the same functionality: counting characters of a string, but implemented in two different ways. Their word-level features are quite different, yet they own the same functionality, which will put a heavy burden on the model training process. So we turn our attention to proposing a new representation of code which not only takes word-level features into account.

Abstract **S**yntax **T**ree (AST) is a tree-structural representation of source code which describes its functionality in a specific programming language. The leaves of the tree usually refer to user-defined values which represent identifiers or variable types in the source code. The non-leaf nodes represent a set of structure in the programming language (such as loops or variable declarations). In Figure 1(a) and Figure 1(b), the lower subfigure is the visualized AST of the above code snippet, respectively.

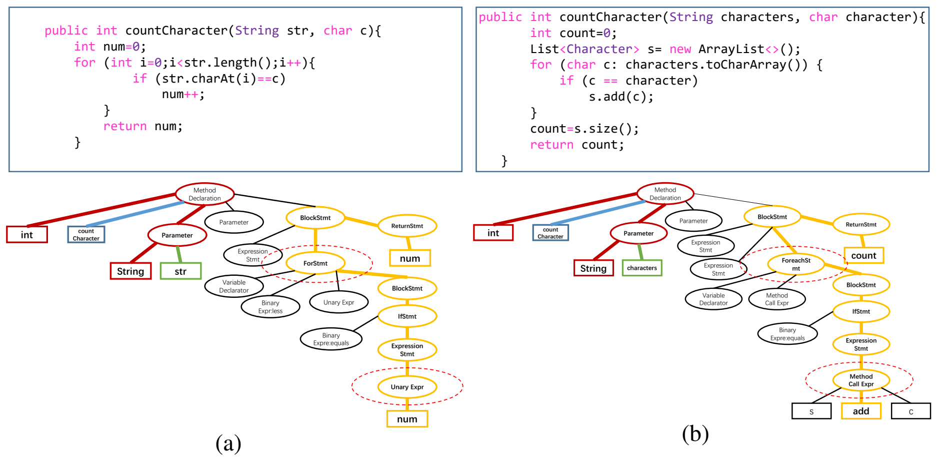
```
public int countCharacter(String str, char c){
    int num=0;
    for (int i=0;i<str.length();i++){
            if (str.charAt(i)==c)
                num++;
        }
        return num;
    }
```
```
public int countCharacter(String characters, char character){
       int count=0;
       List<Character> s= new ArrayList<>();
       for (char c: characters.toCharArray()) {
           if (c == character)
               s.add(c);
       }
       count=s.size();
       return count;
   }
```

Fig. 1: Two Java functions and their ASTs. Two Java functions have the same functionally while implemented in different ways. Although They differ a lot in token-level representation, considering their AST paths, only differ in two nodes, which are circled by red dotted lines

As shown in Figure 1, user-defined identifiers (such as *num*, *str*) and variable types (such as *String*, *int*) are represented as leaves of the tree, and syntactic structure such as judgment statement (*ifStmt*) and loop (*ForStmt*) are represented as non-leaf nodes. In AST, we call a path between two leaves or a leaf and a root an AST path, which are marked red, yellow, blue or green respectively in Figure 1(a) and 1(b). Intuitively, every path is a functional module in the code snippet. It is clear that although two methods are quite different in token-level representation, their AST paths differ only in two nodes, a *ForeachSt* node instead of a *ForStmt* node and a *Method Call Expr* node instead of a *Unary Expr*, which are circled by red dotted lines. So faced with drawback b), we propose a new distributed representation of code which combines AST paths features (i.e., syntactic features) and word-level features (i.e., lexical features) of code in SCE model.

In summary, our contributions in this paper are as follows:

– We propose a new distributed representation of code snippets which combines AST paths features and word-level features of code.
– We propose SCE model. The model uses supervised learning to pre-train distributions of code and NL at the same time, aims to promote aligning them. Based on pre-trained code embeddings, we build a variant of the encoder-decoder model to align NL and code.
– We conducted comparative experiments on real-world datasets including code in popular high-rank Github repositories to evaluate our approaches, and the experiment results indicate that our models are effective and outperform mainstream approaches by 10.15% on performance in code generation.

The rest of this paper is organized as follows: our approach is presented in Section 2. Section 3 introduces the experiments details. Section 4 introduces research related to this work. Then the conclusion is shown in Section 5.

2 Approach

In this section, we introduce the details of our approach. The approach consists of two stages: code embedding and code generation. Figure 2 provides an overview of SCE model for code embedding.

A pair of natural language (*NL*) and code snippet (abbreviated as *Code*) is an input for training, which all consist of a sequence of tokenized tokens[4]. After represented as a vector by one-hot, NL would be fed into NL encoder. The NL encoder is a bidirectional transformer, which captures the contextual information for each word of NL, and generates their contextual embedding vectors $E_1^{nl}, E_2^{nl}, ...E_i^{nl}$. Finally, concatenate them to E^{nl}. Let

$$E_1^{nl}, E_2^{nl}, ...E_i^{nl} = BiTransformer(V_1^{nl}, V_2^{nl}, ...V_i^{nl}) \quad (1)$$

$$E^{nl} = [E_1^{nl}; E_2^{nl}; ...E_i^{nl}] \quad (2)$$

[4] More details about tokenization phase, please refer to Section 3.2

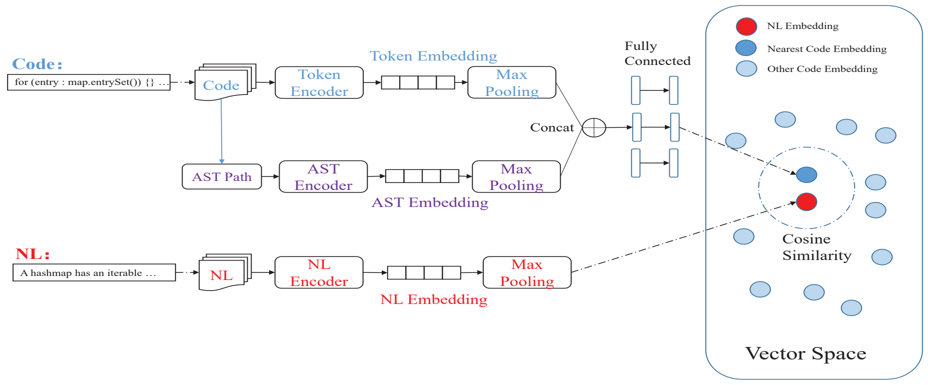

Fig. 2: Overview of Code Embedding Task. Feed NL to NL encoder to get NL embedding E^{nl}, feed code to Token Encoder and code's AST Paths to AST encoder, then combine their output embeddings by a fully connected layer to get E^{code}. Finally, fit the distribution of E^{nl} and E^{code}

where V_{nl}^{i} is denoted as the vector of i_{th} NL token. Then E^{nl} would be fed to a max-pooling layer.

The same as NL representation, we feed code tokens to Token Encoder, which is a bidirectional transformer, the encoder encodes the tokens to Token Embedding vectors and concatenate to E^{tokens}. A max-pooling layer is followed too.

As mentioned in Section 1, we not only capture token-level feature but also capture functionality feature of a code snippet by its AST paths. Every AST path consists of two leaves and non-leaf nodes, so every path is seen as a sequence of its non-leaf nodes' embedding vectors and a sequence of two leaves' tokens embedding vectors. Let

$$V^{nodes} = (V_1^{node}, V_2^{node}, ..., V_i^{node}) \tag{3}$$

$$V^{leaves} = (V_1^{token}, V_2^{token}, ..., V_j^{token}) \tag{4}$$

where V_{node}^{i} is denoted as the vector of i_{th} node. We use bi-direction LSTM to encode the V^{nodes} and V^{leaves}

$$h_1^{node}, h_2^{node}, ...h_i^{node} = BiLSTM(V_1^{node}, V_2^{node}, ..., V_i^{node}) \tag{5}$$

$$h_1^{leaves}, h_2^{leaves}, ...h_j^{leaves} = BiLSTM(V_1^{token}, V_2^{token}, ..., V_j^{token}) \tag{6}$$

and concatenate the bi-direction final hidden states of *LSTM* as the final representation of non-leaf nodes and leaf nodes.

$$E^{nodes}(V_1^{node}, V_2^{node}, ..., V_i^{node}) = [h_i^{node}; h_1^{node}] \tag{7}$$

$$E^{leaves}(V_1^{token}, V_2^{token}, ..., V_j^{token}) = [h_j^{leaves}; h_1^{leaves}] \tag{8}$$

Suppose a snippet of code has k AST paths, we concatenate E^{nodes} and E^{leaves}, then average the combined vector to E^{AST}:

$$E^{AST} = \frac{1}{k}\sum_{k=1}^{k}[E^{nodes}; E^{leaves}] \tag{9}$$

A max-pooling layer is followed to reduce dimensions of E^{AST}.

To represent the final code embedding E^{code}, we concatenate the AST path representation and token-level representation and apply a fully connected layer to combine them

$$E^{code} = tanh(W \cdot [E^{AST}; E^{tokens}]) \tag{10}$$

where W is a $(d_{AST} + d_{tokens}) \times d_{hidden}$ weight matrix.

we choose cosine similarity as our loss function to describe the distance in distribution between E^{nl} and E^{code},

$$D_{loss} = cos(E^{nl}, E^{code}) = \frac{\sum_{i=1}^{n} E_i^{nl} \times E_i^{code}}{\sqrt{\sum_{i=1}^{n} (E_i^{nl})^2} \times \sqrt{\sum_{i=1}^{n} (E_i^{code})^2}} \tag{11}$$

where E_i^{nl}, and E_i^{code} are the i_{th} dimension of E^{nl} and E^{code}.

When SCE model finishes pre-training, we use pre-trained Token encoder to encode our code dictionary to get dictionary embedding matrix E_{dic}.

To evaluate our model, we build a variant of the encode-decoder model with global attention, and it focuses on the stage of generating code from NL with the support of SCE model. Its NL encoder is transferred from SCE model's. Its decoder is a $LSTM$ with global attention and uses embedding matrix E_{dic} to convert code snippets to continuously distributed vectors when in training stage.

3 Experiments

3.1 Datasets

Our collected dataset [5] consists of four open source datasets: Awesome Java[6], CONCODE [8][7], BigCloneBench[8], and JDK source codes. Table 1 shows statistics of our base datasets.

Table 1: Statistics of Datasets

Datasets	**Projects**	**Files**	**Lines**
Awesome java	535	264,284	26,407,592
CONCODE	∼33,000	∼300,000	∼13,000,000
BigCloneBench	10	9,376	2,065,108
JDK	-	7,700	1,009,560

[5] https://drive.google.com/open?id=1nOuZjSS9lUqWfQptUOhfX9kNKd_FeCkn
[6] https://github.com/akullpp/awesome-java
[7] https://drive.google.com/drive/folders/1kC6fe7JgOmEHhVFaXjzOmKeatTJy1I1W
[8] https://github.com/clonebench/BigCloneBench/blob/master/README.md

3.2 Data Preprocessing

For every Java function, function's comments are extracted as NL inputs. The function bodies are treated as our target code to be generated. The NL words and code words are lower-cased. The camel-cased and underline identifiers are split into several words, for example, split *checkJavaFile* to three words: check, Java, File, split *get_user_name* to three words: get, user, name. All punctuation marks are removed. We add *[CLS]* at the beginning of every sentence and add *[SEP]* at the end. *[UNK]* is used to represent words outside the dictionary. After these steps, every word is tokenized to token.

Javaparser lib[9] is used to parse Java source codes. ASTParser lib[10] is used to build AST of code. In order to decrease noise and reinforce the learning process, we only use the first sentence of comments since they already summarize the function of methods according to Javadoc guidance[11]. Some redundant comments, such as empty comments, one-word comments, and non-English comments, are filtered.

Finally, we collect 3,950,164 pairs of $(NL, Code)$ for code embedding, and 1,074,963 pairs of $(NL, Code)$ for code generation. Each dataset is split into a training set, and a test set in proportion with 8:2 after shuffling the pairs.

3.3 Experiment Setting

When in code embedding, we restrict the maximum length of NL to 20 words, and the length of the code is limited to 100. We use 12 hidden layers to encode NL and code tokens in the transformer, and the hidden size is 768. When in code generation, we set the hidden size of the *LSTM* cells to 512, and all cells are 2-layers. Max-pooling layer is used to reduce computation and align matrix. We use dropout with p = 0.4. The ratio of teacher forcing is 0.5. Adam is used as our optimizer with an initial learning rate of 0.0001 for optimization. We use TensorFlow to implement our models.

3.4 Experiment Results

To evaluate the quality of the output, following recent works in code generation [11,8], we choose the **BLEU**, **Precision**, **Recall** and **F-score** of generated words as our metrics. We compare our model with two baselines:

- **Code generated methods proposed by Iyer et al. [8]** This method is proposed recently and outperforms the state-of-the-art in code generation. It is abbreviated as *Iyer et al.*.
- **Code Generation Task without Code Embedding** To evaluate the effect of code embedding, we conducted an experiment that only utilizes code generation task to generate code which is abbreviated as *without Embedding*.

Table 2 illustrates the precision, recall, F-score, and BLEU results of our approach and other baseline methods. Our approach, abbreviated as *ours*, outperforms all baselines in Precision, Recall, F-score and BLEU. Our approach achieves an improvement

[9] https://github.com/javaparser/javaparser
[10] http://help.eclipse.org/mars/index.jsp
[11] https://www.oracle.com/technetwork/articles/java/index-137868.html

Table 2: Precision, Recall, F-score and BLEU of Our Approach and Other Baselines

Approaches	**Precision**	**Recall**	**F-score**	**BLEU**
Iyer et al.	22.67	13.56	16.97	19.35
without embedding	19.35	11.32	14.28	20.50
Ours	**28.58**	**16.36**	**20.81**	**22.58**

of 2.08 BLEU points compared with the best results of other approaches, which outperforms current state-of-the-art methods by 10.15%. The BLEU scores of *ours* and *without Embedding* show that our SCE model could promote the effect of code generation.

4 Related Work

A number of previous researches have explored mapping NL to code blocks [17,9,4], regular expression [12] and SQL statements [18]. Some researches generate code on a certain context: [13,15] generate code in the environment of database querying; Some specific research [10] generates codes in the field of a card game, conditioned on categorical card attributes. These works are based on a chunk of codes that implement certain business logic. Recent researches propose models and evaluate them on domain-specific dataset (Hearthstone & MTG, [10]; CONCODE [8]), and manual labeled per-line comments (DJANGO [14]). Domain-specific data is organized based on specific business logic. Each functionality contains business knowledge and consists of several basic operations. Manually labeled data (DJANGO) contains programs with short description possibly mapping to categorical data. The values need to be copied onto the resulting code from a single domain. Some researchers use AST to represent code[3,2] too, but they don't fuse word-level features.

In Neural Machine Translation area, neural encoder-decoder has proved to be effective. It also has good performance in mapping NL to programming logic and code generation. Some methods directly generate code blocks or domain-specific programming language using encoder-decoder model [14,10]. Some methods use a customized decoder for capturing code structure and perform generation [16].

5 Conclusion

In the paper, we propose a new representation of code snippets which combine features from lexical level and syntactic level. We propose a novel **S**upervised **C**ode **E**mbedding (SCE) model to learn distributed representation of the code and NL at the same time. We conducted several comparative experiments to prove our approach, and experimental results show that our approach, which outperforms state-of-the-art baselines, is significantly effective and can generate more high-quality codes. Our future work will focus on two aspects: how to better fuse other information, and how to generate executable code directly.

References

1. Allamanis, M., Tarlow, D., Gordon, A.D., Wei, Y.: Bimodal modelling of source code and natural language. In: ICML (2015)
2. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019)
3. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL), 40:1–40:29 (Jan 2019). https://doi.org/10.1145/3290353, http://doi.acm.org/10.1145/3290353
4. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: Learning to write programs. arXiv preprint arXiv:1611.01989 (2016)
5. Dieumegard, A., Toom, A., Pantel, M.: Model-based formal specification of a DSL library for a qualified code generator. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012. pp. 61–62 (2012). https://doi.org/10.1145/2428516.2428527, https://doi.org/10.1145/2428516.2428527
6. Glück, R., Lowry, M.R. (eds.): Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3676. Springer (2005). https://doi.org/10.1007/11561347, https://doi.org/10.1007/11561347
7. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Software and System Modeling **9**(3), 375–402 (2010)
8. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 1643–1652 (2018)
9. Liang, P., Jordan, M.I., Klein, D.: Learning dependency-based compositional semantics. Computational Linguistics **39**(2), 389–446 (2013)
10. Ling, W., Grefenstette, E., Hermann, K.M., Kočiskỳ, T., Senior, A., Wang, F., Blunsom, P.: Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744 (2016)
11. Ling, W., Grefenstette, E., Hermann, K.M., Kocisky, T., Senior, A., Wang, F., Blunsom, P.: Latent predictor networks for code generation (2016)
12. Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R.: Neural generation of regular expressions from natural language with minimal domain knowledge. arXiv preprint arXiv:1608.03000 (2016)
13. Manshadi, M.H., Gildea, D., Allen, J.F.: Integrating programming by example and natural language programming. In: AAAI (2013)
14. Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., Nakamura, S.: Learning to generate pseudo-code from source code using statistical machine translation (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. pp. 574–584. IEEE (2015)
15. Quirk, C., Mooney, R., Galley, M.: Language to code: Learning semantic parsers for if-this-then-that recipes. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). vol. 1, pp. 878–888 (2015)
16. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). vol. 1, pp. 440–450 (2017)
17. Zettlemoyer, L.S., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. arXiv preprint arXiv:1207.1420 (2012)
18. Zhong, V., Xiong, C., Socher, R.: Seq2sql: Generating structured queries from natural language using reinforcement learning. arXiv preprint arXiv:1709.00103 (2017)