Abstract: Automatic code generation is to generate the program code of the corresponding language according to the given natural language description. At present, the mainstream methods basically use neural network to encode natural language description, form an abstract syntax tree in the decoder, and then convert the abstract syntax tree into program code. Although the generated code conforms to specific syntax rules, two problems are still ignored: one is the lack of program testing, which is an indispensable step in the process of complete code implementation; Second, it only focuses on whether the generated code meets the abstract syntax requirements, ignoring the more important requirement - functional requirements. This paper constructs a CodeGen test model for integrating code testing, that is, on the basis of generating code that meets the syntax requirements, add program testing steps, integrate program testing information, and iteratively generate code that meets the functional requirements of the program, so as to improve the quality of code generation. This paper evaluates the effect of CodeGen test model on a python data set "hearthstone legend". The experimental results show that this method can effectively improve the quality of generated code. Compared with the existing optimal model, CodeGen test model improves the Bleu value by 0.4%, Rougel value by 0.4%.

Key words: automatic code generation; CodeGen-Test; test information; iteration

I. Introduction

In recent years, with the rapid development of neural machine translation technology and the outstanding performance of sequence-to-sequence models in the field of machine translation, researchers have used neural networks for code auto-generation tasks. However, early methods simply generated sequences of code vocabulary, which did not meet the requirements of correct code syntax. Therefore, some believe that the task of code auto-generation cannot ignore the underlying syntax of the code, and various methods have been proposed. For example, Dong et al.[1] proposed the seq2tree model, which generates code in a multi-layer tree manner to meet the structural requirements of the code; Rabinovich et al.[3] used different decoders based on the generated abstract syntax tree nodes to improve the model's performance; Sun et al.[4,5] employed convolutional neural networks[4] and Transformers[6] to enhance the effect of generating abstract syntax trees.

The current method of generating code using abstract syntax trees has improved the syntactical correctness of the code and has continuously achieved improvements in evaluation metrics such as BLEU[7] scores and accuracy. However, on one hand, achieving syntactic correctness in code is only the first step in code generation tasks. What's more important next is to ensure the functional correctness of the code. On the other hand, current code auto-generation models lack a program testing step, making it difficult to assess the functional correctness of the generated code. To address this issue, this paper introduces a program testing step during the code generation process, quantifies the functional correctness of the generated code, and further guides the model to generate functionally correct code by integrating program testing information.

The contributions of this paper are as follows:

1. We propose to add the program test step and incorporate test information in the process of automatic code generation, then generate code by an iterative manner. This provides a novel idea and direction for future research on automatic code generation.

2. A model that integrates program testing information is proposed, and functionally correct code is generated iteratively.

II. Method

Inspired by the complete code development process, we propose the CodeGen-Test model, which adds program test step in the process of code generation, and further guides the model to generate higher quality code combined with program test information.

The model is an iterative mode, and N represents the number of iterations. In the first round of iteration, the model generates code from the input of natural language description (NL), and then converts the code to abstract syntax tree to obtain rule sequence (Last Code Rule) of the abstract syntax tree. Meanwhile, the generated code is input into the code test process to obtain the Test Information (Test-Info). In the second iteration, input the Test-Info, Last Code Rule and NL obtained in the previous round into the model, generate the code again, and then obtain the Last Code Rule and Test-Info according to the newly generated code. The model performs multiple rounds of iteration. In the first round of iteration, only NL is input, and both Test-Info and Last Code Rule are empty.

The model flowchart is shown in Figure 1. Based on TreeGen[5], this model adds Test-Info Encoder and Code Encoder. The information output by each encoder is integrated in the decoder to generate code by generating abstract syntax
tree. In the generation process, pointer network[16] is used to fill the terminal node of abstract syntax tree.

A. Test-Info Encoder

The fusion of test information helps the model learn the shortcomings of the existing code and improve it in the next generation process. The Test-Info Encoder is responsible for receiving Test Information (Test-Info). The encoder is composed of \( N_t \) blocks, each block contains three different sub layers. Between each two sub layers, residual connection[13] is adopted, and layer normalization[12] is carried out.

Self Attention mechanism can effectively alleviate the problem of long-distance dependence in sequence to sequence tasks. The self attention layer of the model follows the Transformer[6] architecture and uses the method of Dehghani[14] to embed the \( b \)th position of the word in the \( i \)th Transformer block as follows:

\[
p_{b,i}[2j] = \sin \left( \frac{(i + b)j}{(10000^2)} \right)
\]

\[
p_{b,i}[2j + 1] = \cos \left( \frac{(i + b)j}{(10000^2)} \right)
\]

where, \( d \) represents the dimension of word embedding. Then, the latent semantic information is learned through multi head attention to generate a matrix:

\[
Y^{(self)} = \text{concat}(head_1, \cdots, head_h)W_h
\]

where, \( h \) represents the number of multiple heads and \( W_h \) is the weight matrix. The self attention mechanism is applied to \( head_i \), and the calculation equation is as follows:

\[
head_i = \text{softmax} \left( \frac{QK^T}{\sqrt{d}} \right)V
\]

where, \( Q, K, V \) calculated by the following equation,

\[
\begin{bmatrix} Q, K, V \end{bmatrix} = \begin{bmatrix} x_1, \cdots, x_L \end{bmatrix}^T \begin{bmatrix} W_Q, W_K, W_V \end{bmatrix}
\]

(5)

where \( W_Q, W_K, W_V \in \mathbb{R}^{L \times d} \) are the model parameters. \( x_t \) is the input of this coding block, it is the vector sum of word embedding and position embedding in first block, that is \( s_t + p_{L_t} \), and it is the vector sum of the output of the previous coding block and the position embedding of the block in other blocks.

After calculating self attention mechanism, the model adopts gating mechanism to further combine the information embedded in characters.

The vocabulary is represented by character embedding through the full connection layer.

\[
s_i^{(c)} = W^{(c)} \begin{bmatrix} c_1^{(c)}, c_2^{(c)}, \cdots, c_m^{(c)} \end{bmatrix}
\]

(6)

For the \( i \) word, the output of the upper layer \( y_i^{(self)} \) is linearly transformed to obtain the control vector \( q_i \) and \( k_i^{(y)} \).

\[
q_i = W_y y_i^{(self)}
\]

(7)

\[
k_i^{(y)} = W_k y_i^{(self)}
\]

(8)

Obtained \( k_i^{(c)} \) from \( s_i^{(c)} \) in equation (6) by linear transformation.

\[
k_i^{(c)} = W_k \bar{n}_i^{(c)}
\]

(9)

\( \alpha_i^{(y)}, \alpha_i^{(c)} \) are calculated as follows:

\[
\begin{bmatrix} \alpha_i^{(y)}, \alpha_i^{(c)} \end{bmatrix} = \text{softmax} \left\{ q_i^T k_i^{(y)}, q_i^T k_i^{(c)} \right\}
\]

(10)
Similarly, the upper layer feature \( v^{(y)}_i \) and character embedding feature \( v^{(c)}_i \) are obtained from \( y^{(sel)}_i \) and \( n_i^{(c)} \) by linear transformation respectively.

\[
v^{(y)}_i = W^{-} y^{(sel)}_i \tag{11}
\]
\[
v^{(c)}_i = W^{-} n_i^{(c)} \tag{12}
\]

\( \alpha_i^{(y)}, \alpha_i^{(c)} \) for weighted \( v^{(y)}_i \) and \( v^{(c)}_i \), respectively.

The model applies the convolution layer to the output of the gating mechanism layer, to extract the local features around each word. \( y^{(conv)}_i \) is calculated by the following equation:

\[
y^{(conv)}_i = W^{(conv)} [y^{(conv,i-1)}_{i-w}; \ldots; y^{(conv,i-1)}_{i+w}] \tag{13}
\]

where, \( W^{(conv)} \) is the weight of convolution kernel, \( w = \frac{k-1}{2} \), \( k \) represents the size of convolution window.

B. Code Encoder

The code has a strong internal relationship with its test information. In order to enable the model to learn the internal relationship between the code generated in the last round and its corresponding test information.

We convert the code of last round generated to an abstract syntax tree to obtain the rule sequence. Then take rule sequence as input of code encoder. The positional embedding is conducted by equation (1)(2) to learn the positional relationship, and then a self-attention mechanism with different weights from equation (3)(4)(5) is used to obtain the feature vector of codes.

C. Decoder

Decoder is responsible for fusing the information of each part and predicting the current rules of abstract syntax tree. On the basis of TreeGen[5] model decoder, this decoder adds abstract syntax tree attention layer (AST attention) and natural language description attention layer (NL attention), which are respectively responsible for fusing the output of test information encoder and code encoder.

We take the tree path as the input of Decoder. Tree Path is a path from the root node to the currently node that to be expanded in the abstract syntax tree of Generated code.

Then predicted the rule of current nodes that need to expand by full connection layer. In the prediction process, the finger network[11] is also used to fill the terminal nodes of the abstract syntax tree. The model is finally optimized using the default parameters of the AdaFactor optimizer[10].

III. Experiments

A. Datasets

We evaluated our approach on the "HearthStone" benchmark, which is a public benchmark dataset collected from card game. Each card consists of a semi-structured description and a python program. This description is attached with some attributes, such as card name, card type, and natural language description of card function. As shown in Figure 2, the natural language description part of the card and the correct code constitute a sample of the data set.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Target} & \textbf{StrAcc} & \textbf{Acc+} & \textbf{Bleu} & \textbf{RougeL} \\
\hline
LPN & 6.1 & - & 67.1 & - \\
SEQ2TREE & 1.5 & - & 53.4 & - \\
YN17 & 16.2 & 18.2 & 75.8 & - \\
ASN & 18.2 & - & 77.6 & 77.5 \\
ASN+supatt & 22.7 & - & 79.2 & - \\
ReCode & 19.6 & - & 78.4 & - \\
SZM19 & 27.3 & 30.3 & 79.6 & 82.8 \\
TreeGen-A & 25.8 & 25.8 & 79.3 & - \\
TreeGen-B & 31.8 & 33.3 & 80.8 & 82.9 \\
ADGSeq2Seq & 27.3 & - & 78.1 & 87.4 \\
CodeGen-test & 31.8 & 33.3 & 81.2 & 87.8 \\
\hline
\end{tabular}
\caption{Model comparison experiments}
\end{table}
The results show that CodeGen-Test model has the best performance on Bleu, Rouge-L[17] and Test-Acc metric. CodeGen-Test model outperform all methods except TreeGen-B in StrAcc and all methods except TreeGen-B and sszm19 in ACC+ metric. The above experimental results verify the effectiveness of CodeGen-Test in the task of automatic code generation.

In order to verify the performance of the model under different iteration numbers, experiments were conducted with iteration counts N ranging from 1 to 3.

| Target                | N  | StrAcc | acc+ | Bleu  |
|-----------------------|----|--------|------|-------|
| CodeGen-test (full model) | 1  | 27.3   | 28.9 | 78.9  |
|                        | 2  | 30.3   | 33.3 | 81.0  |
|                        | 3  | 31.8   | 33.3 | 81.2  |
| CodeGen-test (-Test Info Encoder) | 1  | 27.3   | 27.3 | 78.9  |
|                        | 2  | 27.3   | 27.3 | 79.7  |
|                        | 3  | 27.3   | 28.9 | 80.1  |
| CodeGen-test (-Code Encoder) | 1  | 22.7   | 22.7 | 77.5  |
|                        | 2  | 25.8   | 25.8 | 79.2  |
|                        | 3  | 27.3   | 28.9 | 80.2  |
| CodeGen-test (-Test Info Encoder -Code Encoder) | 1  | 22.7   | 22.7 | 78.5  |
|                        | 2  | 27.3   | 27.3 | 79.0  |
|                        | 3  | 27.3   | 27.3 | 79.8  |

The experimental results are shown in Table 2. The model performance is improved with the increase of iteration times. The results of the four models in Table 2 are better when iteration times n = 3 than iteration times n = 1; On the other hand, it is also noted that when n=2, the increase of iteration times does not significantly improve the model. In Table 2, only the third and fourth models are improved with the increase of iteration times when iteration times n=2. We think this is partly due to the gradual decrease in the number of data sets during the experiment. With the increase of the number of iterations n, the number of samples with test information content is further reduced, which limits the further improvement of the model.

Overall, the model performance improves with iteration times increase.

IV. Conclusion

This paper proposes an automatic code generation model, CodeGen-Test that integrates program test information. The model is based on sequence to sequence framework. In this paper, we also test the generated code, improve the process of automatic code generation technology, and integrate program test information to guide the model.

In future work, we will incorporate information from the code API to guide the model to achieve further improvements in cases such as function calls, parameter passing, etc.

Acknowledgments

The research is supported by the National Natural Science Foundation (NNSF) of China (No. 61877031) and the Jiangxi Normal University Graduate Innovation Fund (Grant No.YJS2022029). We thank all reviewers for their valuable comments.

References

[1] Dong L, Lapata M. Language to logical form with neural attention. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin, Germany: Association for Computational Linguistics, 2016: 33–43.
[2] Yin P, Neubig G. A syntactic neural model for general-purpose code generation. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vancouver, Canada: Association for Computational Linguistics, 2017: 440–450.
[3] Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. arXiv:1704.07535 [cs, stat], 2017.
[4] Sun Z, Zhu Q, Mou L, Xiong Y, Li G, Zhang L. A grammar-based structural cnn decoder for code generation. Proceedings of the AAAI Conference on Artificial Intelligence, 2019, 33: 7055–7062.
[5] Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L. TreeGen: a tree-based transformer architecture for code generation. AAAI 2020: 8984–8991, 2019.
[6] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. Guyon I, Luxburg U V, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R. Advances in Neural Information Processing Systems. Curran Associates, Inc., 2017, 30.
[7] Papineni K, Roukos S, Ward T, Zhu W-J. Bleu: a method for automatic evaluation of machine translation. Proceedings of the 40th annual meeting of the Association for Computational Linguistics. 2002: 311–318.
[8] Lyu C, Wang R, Zhang H, Zhang H, Hu S. Embedding api dependency graph for neural code generation. Empir. Softw. Eng., 2021, 26(4): 61.
[9] Hayati SA, Olivier R, Avvaru P, Yin P, Tomasic A, Neubig G. Retrieval-based neural code generation. EMNLP 2018: 925–930, 2018.
[10] Shazeer N, Stern M. Adafactor: adaptive learning rates with sublinear memory cost. International Conference on Machine Learning, PMLR, 2018: 4596–4604.
[11] See A, Liu PJ, Manning CD. Get to the point: summarization with pointer-generator networks. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vancouver, Canada: Association for Computational Linguistics, 2017: 1073–1083.
[12] Ba JL, Kiros JR, Hinton GE. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.
[13] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.
[14] Dehghani M, Gouws S, Vinyals O, Uszkoreit J, Kaiser Ł, Lukasz S. Universal transformers. arXiv preprint arXiv:1807.03819, 2018.
[15] Ling W, Grefenstette E, Hermann KM, Köcüksü T, Senior A, Wang F, Blunsom P. Latent predictor networks for code generation. ACL (1) 2016, 2016.
[16] A. See, P. J. Liu, and C. D. Manning, “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). Vancouver, Canada: Association for Computational Linguistics, 2017, pp.1073–1083.[Online].Available:https://www.aclweb.org/anthology/P17-1099.
[17] Lin C-Y, Cao G, Gao J, Nie J-Y. An information-theoretic approach to automatic evaluation of summaries. Proceedings of the Human Language Technology Conference of the NAACL, Main Conference. New York City, USA: Association for Computational Linguistics, 2006: 463–470.