Paper Summary

Süleyman Ateş

May 2022

For UIUC SE Summer Research Program application screening process task, I will be summarizing the paper *CODE2SEQ: GENERATING SEQUENCES FROM STRUCTURED REPRESENTATIONS OF CODE*. Paper can be found via this link.

1 Problem

Generating natural sequences from source code snippets has diverse applications such as code summarization, code captioning and documentation. In this paper, a general problem of generating a natural language sequence from a given source code snippet is considered. For these purposes, sequence-to-sequence (seq2seq) models based on NMT have great performance on these tasks by applying the sequence of tokens method to source code. A new model called **code2seq** is presented as an alternative approach for this problem.

2 Proposed Solution and Differences With Previous Studies

As mentioned earlier, code2seq model is presented as an alternative solution in this paper, and it basically tries to leverage the syntactic structure of programming languages to better encode source code. What makes code2seq impressive and effective is that it uses a different structural encoding method which is representing a code snippet as the set of compositional paths in its abstract syntax tree (AST). After that it uses attention to filter the more relevant paths in decoding part. Let me dive into structural encoding part more.

For a given AST of a code snippet, code2seq considers all pairwise paths between terminals (leaves of AST), and represents them as sequence of nonterminal and terminal nodes. After that, with terminals' values the model use these paths to represent the given code snippet. Thanks to this structural encoding manner, encoder used in code2seq can generalize much better to unseen examples since AST normalizes a lot of surface from variance. AST paths can be represented as a set of k paths: $\{(v_1^1 v_2^1 \cdots v_{l_1}^1), \cdots, (v_1^k v_2^k \cdots v_{l_k}^k)\}$, where l_j is the length of jth path.

2.1 Model Architecture

code2seq follows the standard encoder-decoder architecture for NMT while it differs significantly in the sense that the encoder creates a vector representation for each AST path separately (classical NMT encoders read the input as a flat sequence of tokens). After that, the decoder attends over the encoded AST paths in vector forms while generating the target sequence.

$$x = (x_1, \cdots, x_n) \xrightarrow{encoder} z = (z_1, \cdots, z_n)$$
$$z = (z_1, \cdots, z_n) \xrightarrow{decoder} y = (y_1, \cdots, y_n)$$

Note that the decoder in code2seq uses the average of all of the k paths as the start state. Moreover, at each time step t because of the attention, a context vector c_t is computed by attending over in z using the state h_t which usually computed by LSTM.

$$p(y_1, \cdots, y_m | x_1, \cdots, x_n) = \Pi_{j=1}^{m} p(y_j | y_{<j}, z_1, \cdots, z_n)$$

$$\alpha^t = softmax(h_y W_\alpha z) \qquad\qquad c_t = \sum_i^n \alpha_i^t z_i$$

$$p(y_j | y_{<j}, z_1, \cdots, z_n) = softmax(W_s tanh(W_c[c_t; h_t]))$$

2.2 AST Encoder

As mentioned earlier, from given AST paths $\{x_1, \cdots, x_k\}$, code2seq's encoder creates vector representations z_i's. Here, each path is represented separately by using bi-directional LSTM to encode the path, and sub-token embeddings to get the compositional nature of the terminals' values.

Path Representation

All AST paths include nodes and their child indices from a vocabulary up to 364 symbols together. Each node is represented by a learned matrix E^{nodes}, and then encoded the whole sequence with using the final state of bi-directional LSTM:

$$h_1, \cdots, h_l = LSTM(E_{v_1}^{nodes}, \cdots, E_{v_l}^{nodes})$$

$$\text{encode\_path}(v_1, \cdots, v_l) = [\overrightarrow{h_l}; \overleftarrow{h_1}]$$

Token Representation

As mentioned earlier, the first and the last node of an AST path are terminals having values which are tokens. Moreover, each token is splitted into subtokens e.g. "TreeNode" $\rightarrow$ "Tree" and "Node". A learned matrix $E^{subtokens}$ is used here to represent each subtoken, they are all summed to give the full token:

$$\text{encode\_token}(w) = \textstyle\sum_{s \in split(w)} E_s^{subtokens}$$

Resulting Combined Representation

At the end, path's representation is concatenated with the token representations of each terminal node, and applied a fully-connected layer:

$$z = tanh(W_{in}[encode\_path(v_1, \cdots, v_l); encode\_token(value(v_1)); encode\_token(value(v_1))])$$

where *value(v_l)* gives the corresponding value of a terminal node, and W_{in} is a $(2d_{path} + 2d_{token}) \times d_{hidden}$ matrix.

Lastly, as mentioned earlier, the decoder's start state is the combined representations of all the k paths in the given sample:

$$h_0 = \frac{1}{k} \sum_{i=1}^{k} z_i$$

Important Note: In code2seq, the order of the input random paths is not taken into account unlike typical encode-decoder models.

Attention

At the end, the decoder creates the output sequence while attending over all the $z_1, \cdots, z_k$. Attention is important for selecting the distribution among $z_1, \cdots, z_k$ in decoding.

3 Experimental Results

code2seq is evaluated on two code-to-sequence tasks namely summarization which consists of predicting Java methods' names from their bodies, and code captioning which aims to generate natural language descriptions of C# code snippets.

Model	Java-small			Java-med			Java-large		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
ConvAttention (Allamanis et al., 2016)	50.25	24.62	33.05	60.82	26.75	37.16	60.71	27.60	37.95
Paths+CRFs (Alon et al., 2018)	8.39	5.63	6.74	32.56	20.37	25.06	32.56	20.37	25.06
code2vec (Alon et al., 2019)	18.51	18.74	18.62	38.12	28.31	32.49	48.15	38.40	42.73
2-layer BiLSTM (no token splitting)	32.40	20.40	25.03	48.37	30.29	37.25	58.02	37.73	45.73
2-layer BiLSTM	42.63	29.97	35.20	55.15	41.75	47.52	63.53	48.77	55.18
TreeLSTM (Tai et al., 2015)	40.02	31.84	35.46	53.07	41.69	46.69	60.34	48.27	53.63
Transformer (Vaswani et al., 2017)	38.13	26.70	31.41	50.11	35.01	41.22	59.13	40.58	48.13
code2seq	**50.64**	**37.40**	**43.02**	**61.24**	**47.07**	**53.23**	**64.03**	**55.02**	**59.19**
Absolute gain over BiLSTM	+8.01	+7.43	+7.82	+6.09	+5.32	+5.71	+0.50	+6.25	+4.01

Figure 1: Code Summarization tasks performances for each model. code2seq outperforms related previous models in the field.

Model	BLEU
MOSES† (Koehn et al., 2007)	11.57
IR†	13.66
SUM-NN† (Rush et al., 2015)	19.31
2-layer BiLSTM	19.78
Transformer (Vaswani et al., 2017)	19.68
TreeLSTM (Tai et al., 2015)	20.11
CodeNN† (Iyer et al., 2016)	20.53
code2seq	**23.04**

Figure 2: Code Captioning tasks performances for each model. code2seq outperforms related previous models in the field.

4 Ablation Study

To capture the contributions of different parts of code2seq architecture, an ablation study is conducted.

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

Figure 3: Variations of code2seq model to better understand the effects of each component.

What is essential and what makes code2seq successful is that the focus it gives to AST leaves. Thanks to this, the model can ignore unimportant details, e.g. semicolons, brackets etc., and can focus more on named tokens to construct better connections in these tasks. It can be seen in Figure 3 in *No AST nodes (only tokens)* row. As a result, this structural representation power make code2seq a state-of-the-art and uniquely different work in the field of generating natural sequences from source code snippets.