IJCNN 2019. International Joint Conference on Neural Networks. Budapest, Hungary. 14-19 July 2019

An Improved Recurrent Neural Network Language Model for Programming Language

Liwei Wu
Software Institute
Nanjing University
Nanjing, China
mg1632006@smail.nju.edu.cn

Youhua Wu
Software Institute
Nanjing University
Nanjing, China
mg1632007@smail.nju.edu.cn

Fei Li
Software Institute
Nanjing University
Nanjing, China
mg1732005@smail.nju.edu.cn

Tao Zheng
Software Institute
Nanjing University
Nanjing, China
zt@nju.edu.cn

***Abstract*—Language models are applied to the programming language. However, the existing language models may be confused with different tokens with the same name in different scopes and can generate the syntax error code. In this paper, we proposed a grammar language model to solve these two problems. The model is an improved recurrent neural network language model. The improved recurrent neural network language model has the scope-awared input feature and the grammar output mask. We evaluated our model and existing language models on a C99 code dataset. Our model gets a perplexity value of 2.91 and a top-1 accuracy rate of 74.23% which is much better than other models.**

***Index Terms*—Programming Language,Language Model,Deep Learning**

I. Introduction

Recently, natural language processing(NLP) methods have been applied on the software engineering, such as code completion [1], [2], language migration [3], code synthesis [4], code summarization [5], [6] and so on.

Language model(LM) is one of the most widely-used NLP methods applied to programming language. A language model is a probability distribution over sequences of words. Given a sequence whose length is n, it will assign probability $P(x_1, x_2, ..., x_n)$ to the sequence. For compute this probability, a decomposition is applied to the target probability $P(x_1, x_2, ..., x_n) = \prod_{i=1}^{i=n} P(x_i \mid x_1, ..., x_{i-1})$, so language model needs to compute the distribution of $P(x_t \mid x_1, ..., x_{t-1})$ where $x_1, ..., x_{t-1}$ is the previous context and x_t is a token from a vocabulary D. The recent best language model is the neural network language model [7], [8]. In NLP, neural network language model has been shown to be able to learn the high-level structure of the language [9]. The conditional recurrent language model has become the standard decoder of the sequence to sequence model [10].

Applying Neural Network Language Model(NNLM) to the programming language is straightforward since we can simply take the code as a sequence of lexical tokens [11]. However, there are two problems in this view. Firstly, this model cannot distinguish two different tokens with the same name but in different scopes since the NNLM only takes the code as the lexical token sequence. For example, in Fig. 1a, there are two pairs of tokens with the name x and the name y in the main scope and the minus scope. If we take the code as a sequence of lexical tokens, the tokens with the same name will be the same token. However, we know that in this code the token x in the main scope is corresponding to the token y in the minus scope. Secondly, the grammar in programming language is well defined and more important than the grammar in natural language. A minor grammar error in programming language can make the code compile failed. NNLM always gives the grammar error code a positive probability whose probability should be zero. For example, in the Fig. 1b, the penultimate right brace is lost. The probability of this code should be zero, but NNLM gives it a positive probability.

To solve the above problems, we proposed a grammar language model. The grammar language model is a recurrent language model with two new techniques – the scope-awared input feature and the grammar output mask. We maintain a scope stack. The stack contains feature vectors for each scope. A scope-awared token feature vector is calculated from the original token embedding feature and the corresponding scope feature vector. The scope-awared token feature vector is the input of our grammar language model. The grammar of most programming languages is LL(k) grammar. A parse table can be generated from the LL(k) grammar. When parsing a token sequence, the lookahead tokens should be in the parse table key set if and only if the token sequence are syntax right, so we can create a grammar output mask to rule out all syntax error tokens from the LM output distribution.

We trained our model on a C dataset collected from the CodeForces which is an online judge website. The grammar language model gets a perplexity value of 2.91 and a top-1 accuracy rate of 74.23%.

II. Related work

Statistical language models are used in many natural language technologies. N-gram models are widely used in statistical natural language processing due to their simplicity and good modeling performance, such as automatic speech recognition (ASR) [12], machine translation [13], [14], and spelling correction [15]–[17].

Recently, neural network has been successfully applied to natural language processing. Xu *et al.* [18] firstly used artificial neural networks to learn the language model, which is proved to have better prerformence than standard statistical methods.

978-1-7281-2009-6/$31.00 paper N-19237.pdf

```
int minus(int x, int y)
{
    int b=2;
    return b+x-y;
}

int main()
{
    int x=0, y=4;
    printf("%d", minus(y, x));
}
```

(a)

```
#include <stdio.h>
int main()
{
  int i = 0, sum = 0;
  for(i=0;i<10;i++)
  {
    sum += i;

  printf("sum=%d", sum);
}
```

(b)

Fig. 1: Sample Code

Bengio *et al.* [19] generated a distributed representation of words along with the probability function for word sequences. A sequence of words gets high probability if the words have a nearby representation. Recurrent neural network language models (RNNLMs) have recently demonstrated state-of-the-art performance across a variety of tasks. Mikolov *et al.* [7], [8] created and improved recurrent neural network based language model (RNNLM) using a hidden layer to store data context. Context can cycle inside these networks for arbitrarily long time. Bengio *et al.* [20] showed that learning long-term dependencies by stochastic gradient descent can be quite difficult. Arisoy *et al.* [21] proposed deep-neural-networks-based language models (DNN LMs) to capture higher-level discriminative information about input feature. In addition to n-gram model, Sutskever *et al.* [10] presented a general end-to-end approach to sequence learning. Sequence-to-sequence model contains two multilayered RNNs. One encodes the input sequence to a vector of a fixed dimensionality and the other decodes the target sequence from the vector. Some language and translation models have added the soft attention or memory mechanisms [22]–[24]. The pointer mechanism is shown to be helpful in tasks like summarization [25], neural machine translation [26], code generation [27] and language modeling [28].

Because of the good performance in natural language technologies, language models are also used in code completion and synthesis. Hindle *et al.* [2] used n-gram model Manning *et al.* [29] on lexical tokens to show that source code has repetition and predictability. Source code can be usefully captured by language models originally developed in the field of statistical natural language processing (NLP). Much papers combine different features of program with language model to solve code completion. Tu *et al.* [30] improved n-gram model with caching capability for recently seen tokens. Raychev *et al.* [31] captured common sequences of API calls with n-gram to predict API call. Nguyen *et al.* [1], [32] incorporated syntactic and semantic information into code tokens to increase accuracy of code suggestion. Bhoopchand *et al.* [33] further proposed a special mechanism called sparse pointer for better predicting out-of-vocabulary words in code completion. To solve the gradient vanishing problem, Li *et al.* [34] applied attention mechanism [22] to deal with long-range dependencies. Probabilistic grammars are also employed for code completion [35]–[37]. Bielik generalized probabilistic context free grammars (PCFGs) by allowing conditioning of a production rule beyond the parent nonterminal to capture rich contexts relevant to programs. Raychev improved the work of Bielik by learning a decision tree over abstract syntax trees to condition the prediction of a program element.

III. Grammar language model

In natural language, recurrent neural network language model(RNNLM) [8] is one of the most widely-used language model. However, in programming language, RNNLM has two problems as discussed in the Section I. We proposed a grammar language model to solve these problems. In this section, we introduce the global architecture of the grammar language model and then describe the scope-awared and the grammar output mask input in detail.

A. Global architecture

The global architecture is shown in the Fig. 2. The skeleton of our model is RNNLM. For each token x_t, the corresponding input I_t^r is computed as:

$$I_t^r = \begin{cases} I_t^s, & \text{if } x_t \text{ is an identifier} \\ E[x_t], & \text{else} \end{cases} \tag{1}$$

where $E \in \mathbb{R}^{V_v \times M}$ is the embedding look up table, so $E[x_t] \in \mathbb{R}^M$ is its corresponding embedding vector;V_v is the vocabulary size;M is the embedding size; $I_t^s \in \mathbb{R}^M$ is the scope-awared feature. This is shown in the Fig. 2 by the solid line and the dashed line. The solid line means the choose of the scope-awared feature or the choose of the embedding vector. The details of calculating the scope-awared feature are described in the Section III-B. Then this feature will be used as the input of RNN cell. In this model, we choose the LSTM [38], so the RNN output is computed as:

$$h_t, c_t = LSTM(I_t^s, h_{t-1}, c_{t_1}) \tag{2}$$

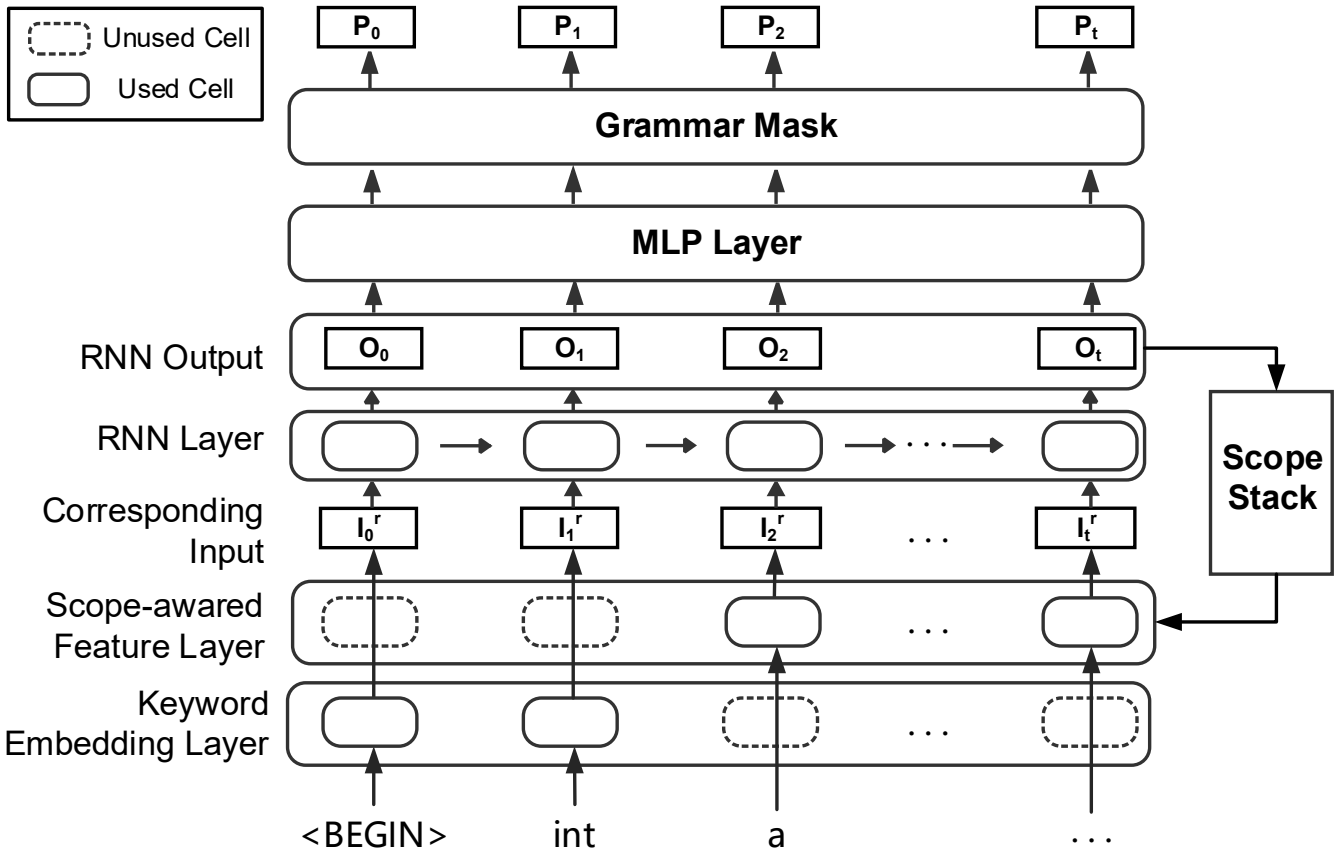

Fig. 2: Global Architecture of Grammar Language Model

where $h_t \in \mathbb{R}^H$ and $h_{t-1} \in \mathbb{R}^H$ are the hidden state; $c_t \in \mathbb{R}^H$ and $c_{t_1} \in \mathbb{R}^H$ are the cell state. We set $o_t = MLP_o([h_t; c_t]) \in \mathbb{R}^H$ as the output of the index t. The output o_t is used to update the scope information stack represented as the blue arrow in the Fig. 2. The update process is described in the Section III-B in detail. Then the conditional probability $P(x_t \mid x_1, ..., x_{t-1})$ is calculated as:

$$logit_t = MLP(o_t) \tag{3}$$

$$x_{1:t-1} = x_1, ..., x_{t-1} \tag{4}$$

$$P(x_t = j \mid x_{1:t-1}) = \frac{Mask(x_t \mid x_{1:t-1})_j e^{logit_t[j]}}{\sum_{i=1}^{V_v} Mask(x_t \mid x_{1:t-1})_i e^{logit_t[i]}} \tag{5}$$

where $Mask(x_t \mid x_1, ..., x_{t-1})$ is the grammar mask to rule out all syntax error tokens which will be described in Section III-C in detail; V_v is the size of the token vocabulary; $logit_t \in \mathbb{R}^{V_v}$ is the logit of the conditional distribution; MLP is a multilayer feedforward neural network. The loss function is the cross-entropy calculated from the conditional probability $P(x_t = j \mid x_1, ..., x_{t-1})$.

B. Scope-awared input

Since the traditional language model is confused with identifier tokens in different scopes with the same name, we can add the scope information to the input to distinguish the different identifier tokens. To maintain the scope information, we maintain a scope information stack S_s which contains all scope vectors.

For every identifier token input x_t, its scope-awared input I_t^s is computed as:

$$I_t^s = MLP([E[x_t]; V_s[y_t]]) \tag{6}$$

where $E \in \mathbb{R}^{V_v \times M}$ is the embedding look up table, so $E[x_t] \in \mathbb{R}^M$ is its corresponding embedding vector; y_t is the index of the scope in which the token x_t is declared, so $V_s[y_t] \in \mathbb{R}^H$ is its corresponding scope vector; MLP is a multilayer feedforward neural network, so $MLP([E[x_t]; V_s[y_t]])$ mixes the token information and the corresponding scope information. The out-of-vocabulary identifier is replaced by a special token —- "<unk>".

The scope information is composed of all the tokens in it, so we can calculate the scope vector $V_s \in \mathbb{R}^H$ token by token. When a scope ends, the top scope vector in the scope information stack should be popped out of the stack. When a scope begins, an empty scope information vector should be pushed into the stack. Initially, the scope vector is zero or sampled from a standard normal distribution. For every RNN

output o_t, we calculate an update value $u_t \in \mathbb{R}^H$ and an update gate $g_t \in \mathbb{R}^H$. The scope vector V_s is updated by this function:

$$V_s := V_s * (1 - g_t) + g_t * u_t \tag{7}$$

The update value u_t is calculated as:

$$u_t = tanh(W_u[V_s; o_t] + b_u) \tag{8}$$

where $W_u \in \mathbb{R}^{H \times 2H}$ and $b_u \in \mathbb{H}$ are the parameters. The update gate g_t is calculated as:

$$g_t = sigmoid(W_g[V_s; o_t] + b_g) \tag{9}$$

where $W_g \in \mathbb{R}^{H \times 2H}$ and $b_g \in \mathbb{H}$ are the parameters. Since a token can be in multiple nested scopes, the update process should be applied to all scope vectors in the scope stack.

C. Grammar output mask

a) Context-Free grammar: A context-free grammar(CFG) [39] is traditionally defined as a 4-tuple $G = (V, \Sigma, R, S)$: V is a finite set of nonterminal symbols; Σ is a finite set of terminal symbols, disjoint from V; R is a finite set of production rules and S is a distinct nonterminal known as the start symbol. The rules R are formally described as $\alpha \to \beta$ for $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ with * denoting the Kleene closure.

Application of a production rule to a nonterminal symbol defines a tree, with symbols on the right-hand side of the production rule becoming child nodes for the left-hand side parent. The grammar G defines all possible tree extending for each nonterminal symbol in V. The tree is constructed by recursively applying productions in R to the nonterminal symbol until all symbols in the tree is in the Σ. The root of the parse tree is the start symbol S. The language of a CFG is the set of the sequence of left-to-right terminal symbols in a right tree. A string in the CFG can be parsed top-down to get a parse tree rooted from the starting symbol S. In programming language, every right code can be parsed to a parse tree.

b) LL(k) Grammar: The LL(k) [40] languages comprise the largest class of context-free languages that can be parsed deterministically from the top down. For every LL(k) grammar, we can generate a parse table. When the table is generated, a table driven parse algorithm can be used to parse a string to get the corresponding parse tree. The algorithm parses the string from left to right and constructs a leftmost derivation of the string. To determine which production should be applied to the current nonterminal symbol, the algorithm looks forward to k tokens at most and looks up the parse table. If the tokens are in the parse table, the corresponding production will be applied. Otherwise, the string is syntax error. It means that a string is syntax right if and only if when parsing it, all lookahead terminal symbols are the prefix of one key in the parse table, so when we have seen the previous $t-1$ tokens, a nonterminal symbol N can be calculated, and meanwhile the grammar output mask can be calculated as:

$$Mask(x_t \mid x_1, ..., x_{t-1}) = \begin{cases} 1, & f(L, T(x_t))) \\ 0, & \text{else} \end{cases} \tag{10}$$

TABLE I: The sample LL(2) grammar

Production	Corresponding Lookahead tokens
A $\to$ aBaa	a
A $\to$ bBba	b
B $\to$ b	bb
B $\to$	ba

where L represents the previously lookahead terminal symbols;T is a function mapping a token to its corresponding terminal symbol;f is a function judging whether $(L, T(x_t))$ is the prefix of one key in the $ParseTable[N]$. The detailed algorithm is show in the Algorithm 1. A sample grammar is shown in the Table I which is a LL(2) grammar. In the Table I the string in the parenthesis is the lookahead key for the corresponding production. If we predict the next token with the prefix "b", the process of executing the Algorithm. 1 is shown in the Table II. Firstly, we need to parse the existing prefix "b" and the start symbol is "A". This is the first line of the Table II. Secondly, as shown in the second line of the Table II, "b" is the lookahead symbol, so based on the second line of the grammar in the Table I the production $A \to bBba$ is applied to the start symbol "A" whose result is show in the third line of the Table II. Thirdly, the top of the stack and the head of the lookahead list are the same, so they will be popped together as shown in the fourth line of the Table II. Finally, the lookahead list is empty and the first symbol in stack is "B", so the next token must be "b" and the mask is $[0, 1]$ since the right lookahead keys of "B" are bb and ba.

IV. Empirical Experiment

a) Dataset: The C dataset is crawled from the CodeForces which contains 97547 codes in the training set, 12140 codes in the validation set and 13791 codes in the test set. The max length of all codes in the dataset is 500. There are not two codes in any two of these three set whose authors are same. All codes in the dataset are C99 standard. Another test dataset is the DeepFix [41] dataset which contains 46500 correct C99 codes. This dataset was collected from a MOOC course. The distribution of this dataset is different from our train dataset. We will evaluate our model on the DeepFix dataset to show the generalization of our model. All comments in the codes are removed.

b) Implementation detail: The word embedding vectors are randomized from a normal distribution and optimized along the training process. The size of embedding vectors is 100. All RNNs and FFNNs in the model are 3 layers' and 2 layers' respectively. The hidden state size is set to 100 for all layers. The model is trained by the SGD optimizer [42] with an initial learning rate of 0.01. The learning rate is scheduled by the ReduceLROnPlateau scheduler of pytorch [43]. The C99 parse table is generated by the SLK Parser Generator[1].

[1]http://www.slkpg2.com/

Algorithm 1: calculate $Mask(x_t \mid x_1, ..., x_{t-1})$

Input: $ParseTable$, $Context = [T(x_1), ..., T(x_{t-1})]$), the token list $Tokens$
Initialize: $stack = [S], i = 0, Lookahead = []$;
while *True* **do**
  $t = stack.top()$;
  if t *is terminal symbol* **then**
    $Lookahead = Lookahead[1:]$;
  else
    if $Lookahead$ *is a key in* $ParseTable[t]$ **then**
      $stack.pop()$;
      append the right side of the production $ParseTable[t][Lookahead]$ to the $stack$;
    else if $Lookahead$ *is the prefix of some keys in* $ParseTable[t]$ **then**
      $i+ = 1$;
      if $i >= t$ **then**
        break;
      else
        $Lookahead.append(Context[i])$;
      end
    else
      The $context$ is not syntax right, **Abort**;
    end
  end
end
$t = stack.top(), mask = []$;
get the set K of all the keys in $ParseTable[t]$ with prefix $Lookahead$;
foreach $token$ *in* $Tokens$ **do**
  if $\exists key \in K, key[len(Lookahead)] == token$ **then**
    $mask.append(1)$;
  else
    $mask.append(0)$;
  end
end
Result: res

TABLE II: The process of Algorithm1 applied on the sample grammar

Matched	Stack	Input	Lookahead tokens	Legal lookahead tokens	Action
	A	b		{a, b}	
	A	b	b	{a, b}	Lookahead the first token.
	bBba	b	b	{b}	Use the production A → bBba.
b	Bba			{ba, bb}	Match the label b. Now the legal lookahead tokens are ba or bb, so the only legal next token is b.

c) Evaluation metrics: Two metrics are used to assess the performance of the four models.

- **Perplexity.** Perplexity [44] is the traditional evaluation metrics in natural language model which measures how well the language model matches the dataset.
- **Top-K accuracy.** Top-K accuracy measures whether the top-K predictions of the conditional probability $P(x_t = j \mid x_1, ..., x_{t-1})$ contain the true token.

d) Baseline: We chose three traditional language model as our baseline – n-gram [2], neural n-gram [21], RNNLM [8]. All hidden state size and embedding size in these three models are set to 100 which is the same as the grammar language model. The n value of the n-gram model is set from 2 to 4 and the neural n-gram is set from 2 to 5.

e) Ablation test: We did ablation test on the grammar language model. We trained a grammar language model without the scope input feature and a grammar language model without the grammar mask output.

f) Result: After trying two different scope vector initialization method – zero vector or sampling from a normal

TABLE III: The Result of Grammar Language Model, baseline and ablation test

MODEL	PERPLEXITY	TOP-1 ACCURACY	TOP-4 ACCURACY	TOP-8 ACCURACY	TOP-12 ACCURACY
2-GRAM	16.28	37.52%	68.89%	80.85%	85.71%
3-GRAM	21.01	52.20%	77.93%	85.61%	88.46%
4-GRAM	47.51	58.65%	79.53%	85.02%	87.19%
NEURAL 2-GRAM	7.92	51.29%	77.43%	85.50%	88.60%
NEURAL 3-GRAM	6.42	57.78%	80.31%	86.78%	89.41%
NEURAL 4-GRAM	5.82	60.88%	81.49%	87.35%	89.79%
NEURAL 5-GRAM	5.41	62.79%	82.85%	88.17%	90.36%
RNNLM	5.07	65.30%	84.00%	88.78%	90.72%
GRAMMAR LANGUAGE MODEL - SCOPE INPUT FEATURE	3.22	72.62%	89.57%	93.29%	94.64%
GRAMMAR LANGUAGE MODEL - GRAMMAR OUTPUT MASK	3.21	72.53%	89.71%	93.34%	94.66%
GRAMMAR LANGUAGE MODEL	2.91	74.23%	90.85%	94.40%	95.60%
GRAMMAR LANGUAGE MODEL(WITH RANDOM INITIALIZER)	2.97	74.16%	90.55%	94.08%	96.32%
GRAMMAR LANGUAGE MODEL(ON DEEPFIX DATASET)	2.90	73.92%	91.65%	95.08%	96.13%

distribution, we found that the initialization method almost does not affect the result of our model. We chose the zero initialization method in our model since it is simpler. The Grammar Language Model gets a perplexity score of 2.91 and a top-1 accuracy score of 74.23% on the test dataset. The detailed result is show in the Table III. It can be seen that our model gets the best result measured by all measure metrics compared to the baseline models. Compared to the result on the original test dataset, the result on the DeepFix dataset is almost the same. This shows that our model has good generalization. From the table, the ablation test result shows that the two techniques proposed in this paper is useful. Replacing the scoped-awared input feature with the traditional embedding lookup input increases the perplexity score by about 0.31 and decreases the top-1 accuracy score by about 1.61%. Replacing the grammar mask output with the traditional softmax output increases the perplexity score by about 0.30 and decreases the top-1 accuracy score by about 1.70%. A line chart of the top-K accuracy changed by the increasing of K value from 1 to 14 is shown in the Fig. 3. In this figure, the best model of n-gram and the best model of neural n-gram models are drawn.

g) Illustrative Examples: We compared the results of two predictions of the sample code shown in Fig. 1a between RNNLM and out model.

- When predicting the underlined identifier “y”, the top-5 predictions of RNNLM are [“n”, “CONSTANT”, “i”, “a”, “x”] with the probability of [0.17, 0.14, 0.08, 0.07, 0.07]. The RNNLM is not very confident of any tokens and the first three identifiers are the identifier declared in the code. However, in our model, the top-5 predictions are [“)”, “x”, “CONSTANT”, “y”, “(”] with the probability of [0.56, 0.27, 0.08, 0.05, 0.01]. Our predictions are also wrong, but they are all syntax right and all identifiers appearing in them are declared in this scope. This means our scope-awared input feature is useful to collect the scope information.

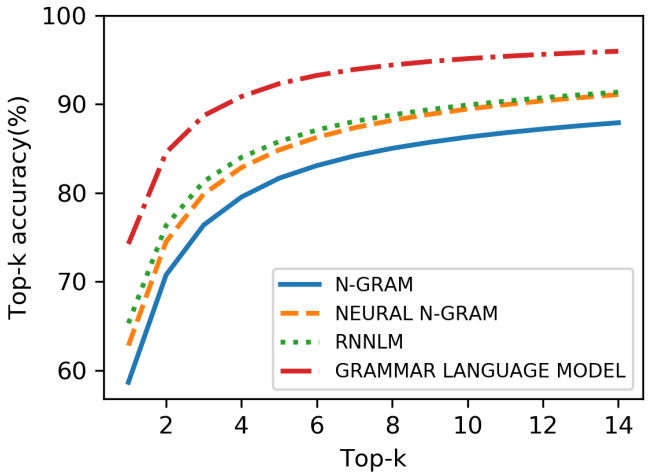

Fig. 3: Top-K accuracy of different approaches

- When predicting the underlined right bracket, the top-5 predictions of RNNLM are [“;”, “==”, “)”, “#”, “%”] with the probability of [0.71, 0.05, 0.04, 0.03, 0.02]. In RNNLM, it is very confident with the top-1 prediction, but the “;” used here will cause a syntax error. However, in our model, the top-5 predictions are [“)”, “,”, “+”, “/”, “*”] with the probability of [0.87, 0.06, 0.02, 0.02, 0.01]. Our predictions are all syntax right and our model is very confident with the true target “)”. This means the grammar output mask is useful.

V. CONCLUSION

In this paper, we proposed a grammar language model for programming language. Different from the previous models, our model utilizes the grammar information to distinguish the different tokens with the same name in different scopes and rules out the syntax error tokens from the prediction. The model is based on the traditional RNNLM, so we believe our model can be used in any place where the RNNLM works.

REFERENCES

[1] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 532–542.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[3] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 585–596.

[4] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 357–367.

[5] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *International Conference on Machine Learning*, 2015, pp. 2123–2132.

[6] L. Guerrouj, D. Bourque, and P. C. Rigby, "Leveraging informal documentation to summarize classes and methods in context," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 639–642.

[7] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ, "Strategies for training large scale neural network language models," in *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*. IEEE, 2011, pp. 196–201.

[8] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[9] H.-S. Le, I. Oparin, A. Allauzen, J.-L. Gauvain, and F. Yvon, "Structured output layer neural network language model," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 5524–5527.

[10] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[11] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, J. N. Amaral, E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, J. N. Amaral *et al.*, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018)*, vol. 29, 2016, pp. 1–5.

[12] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[14] G. Doddington, "Automatic evaluation of machine translation quality using n-gram co-occurrence statistics," in *Proceedings of the second international conference on Human Language Technology Research*. Morgan Kaufmann Publishers Inc., 2002, pp. 138–145.

[15] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," *Ann arbor mi*, vol. 48113, no. 2, pp. 161–175, 1994.

[16] S. Bergsma, D. Lin, and R. Goebel, "Web-scale n-gram models for lexical disambiguation." in *IJCAI*, vol. 9, 2009, pp. 1507–1512.

[17] K. Kukich, "Techniques for automatically correcting words in text," *Acm Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 377–439, 1992.

[18] W. Xu and A. I. Rudnicky, "Can artificial neural networks learn language models?" 2000.

[21] E. Arisoy, T. N. Sainath, B. Kingsbury, and B. Ramabhadran, "Deep neural network language models," in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*. Association for Computational Linguistics, 2012, pp. 20–28.

[19] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[20] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[22] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[23] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," *arXiv preprint arXiv:1601.06733*, 2016.

[24] K. Tran, A. Bisazza, and C. Monz, "Recurrent memory networks for language modeling," *arXiv preprint arXiv:1601.01272*, 2016.

[25] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," *arXiv preprint arXiv:1603.06393*, 2016.

[26] C. Gulcehre, S. Ahn, R. Nallapati, B. Zhou, and Y. Bengio, "Pointing the unknown words," *arXiv preprint arXiv:1603.08148*, 2016.

[27] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[28] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[29] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[30] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[31] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[32] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 323–334.

[33] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *arXiv preprint arXiv:1611.08307*, 2016.

[34] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[35] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 472–483.

[36] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *International Conference on Machine Learning*, 2016, pp. 2933–2942.

[37] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 731–747.

[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[39] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison Wesley*, vol. 7, no. 8, p. 9, 1986.

[40] A. V. Aho and J. D. Ullman, *The theory of parsing, translation, and compiling*. Prentice-Hall Englewood Cliffs, NJ, 1972, vol. 1.

[41] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning." in *AAAI*, 2017, pp. 1345–1351.

[42] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[43] P. C. Team, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," 2017.

[44] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *IEEE transactions on acoustics, speech, and signal processing*, vol. 35, no. 3, pp. 400–401, 1987.