Improve Language Modelling for Code Completion through Learning General Token Repetition of Source Code

Yixiao Yang
School of Software
Tsinghua University
Beijing, China
yangyixiaofirst@163.com

Xiang Chen
Beijing, China
kuailezhish@gmail.com

Jiaguang Sun
School of Software
Tsinghua University
Beijing, China

***Abstract*—In last few years, to solve the problem of code completion, using a language model such as LSTM to learn code token sequences is the state-of-art method. However, tokens in source code are more repetitive than words in natural languages. For example, once a variable is declared in a program, it may be used many times. Other elements such as generic types in templates also occur repeatedly. It is important to capture token repetition of code. For example, if usage patterns of variables are not captured, there is little chance for a model trained on one project to predict the name of an unseen variable in another project correctly. Capturing token repetition of source code is challenging because not only the repeated token but also the place at where the repetition should happen must be both decided at the same time. Hence, we propose a novel deep neural model named REP to capture the general token repetition of source code. The repetitions of code tokens are modeled as edges connecting between repeated tokens on a graph. The REP model is essentially a deep neural graph generation model. The experiments indicate that the proposed model outperforms state-of-arts in code completion.**

***Index Terms*—language model, code completion, code recommendation, code token repetition, deep neural graph generation**

I. INTRODUCTION

In the past few years, language models have attracted the attentions of researchers and achieved a great progress in natural language processing tasks, such as machine translation [1] and text generation [2]. A statistical language model is a probability distribution over sequences of linguistic units such as characters or words. With the rapid growth of open-source code and the high-speed development of artificial intelligence, applying natural language processing techniques to source code has become a research direction. The problem of code completion is difficult and attracts a lot of attentions of researchers in the field of software engineering. Based on the already written code, the system of code completion can recommend the suitable code automatically for software developers to improve the efficiency of software development. Different from code synthesis based on natural languages [3], [4] which requires an explicit intention of code described in natural languages, code completion is aimed at mining and learning the hidden intention of code to recommend suitable code automatically. Due to the predictability [5] of source code, taking source code as natural languages to build language models (n-gram, RNN or LSTM) to suggest code has made great progress in the field of code completion [6]–[11]. Among the previous works, deep learning models such as RNN or LSTM have been proved to be effective. In more subdivided fields of code completion, API usage pattern learning also attracts attention of a large amount of researchers [12], [13]. There are many differences between natural languages and source code. Source code is more regular than natural languages. There exist elements which may occur more than once in one fragment of source code, for example, variables or generic types in templates. Thus, taking source code as natural languages directly is not enough.

DOI reference number: 10.18293/SEKE2019-056

Based on the observation that elements of source code are highly repetitive, a new direction has been opened to capture the token repetition of source code to improve the performance of code completion. When predicting code, if an variable is unseen before, there is little chance to predict that variable correctly. If the usage pattern of an variable has been learned, the place at where the variable will be occurred next time can be decided. Then at next time, the name of that unseen variable can be predicted correctly by copying the name of that previously existed variable at the right place. Other elements such as templates also have the property of token repetition. For example, in Java languages, it is a common scenario to get the element from $ArrayList\langle T\rangle$ and the following code: $ArrayList\langle T\rangle\ arr = ...initialization...;\ T\ t\ =\ arr.get(0);$ is common in many programs. Note that T could be the name of any class. In a new project, if some strange class name such as $UnseenStrange$ is in placed of T and $ArrayList\langle T\rangle$ becomes $ArrayList\langle UnseenStrange\rangle$, through learning the repetitive patterns of source code, the right code: $UnseenStrange\ t\ =\ arr.get(0);$ can be generated.

It is difficult to learn the token repetition of source code. There are two main challenges. The first challenge is that when predicting next token, we need to judge whether the next token should be the repetition of some previously existed token. The second challenge is that if the next token is decided to be the repetition of some previously existed token, we need to decide which previously existed token should be repeated. If a program is huge and contains a large amount of tokens, it is hard to decide which token should be repeated. To address the two challenges, we propose a novel REP model to learn the token repetition of source code. The source code is parsed into a token sequence. This token sequence can be viewed as a linear graph as every token has an hidden incoming edge from its previous token. If two tokens in a token sequence are same, an edge is added between the repeated tokens. Then, a complex graph can be generated based on a token sequence. The extra added edge indicates the repetition of tokens. The task of REP model is to learn the edge connection on the generated graph. The experiments show that the proposed model outperforms state-of-art baselines. In summary, the contributions of this paper include:

1) A novel REP model is proposed to capture the general token repetition of source code;
2) Evaluations on four data sets (total 29.15MB) indicate that the proposed model outperforms the state-of-arts.

II. RELATED WORK

Models for code completion. The statistical language models have been widely used in capturing patterns of source code to solve the problem of code completion. In [5], source code was parsed into lexical tokens and the n-gram model was applied directly to suggest the next lexical token. In [14], a large scale experiments was conducted by using n-gram model and a visualization tool was provided to inspect the performance of the language model for the task of code completion. In SLAMC [6], based on basic n-gram model, associating code lexical tokens with roles, data types and topics was one way to improve the prediction accuracy. Cacheca [7] improved n-gram model by caching the recently encountered tokens in local files to improve the performance of basic n-gram model. Decision tree learning was applied to code suggestion, based on this, a decision tree model which integrates the basic n-gram [10] was proposed for source code. The work [15] abstracted source code into DSL and kept sampling and validating on that specially designed DSL until the good code suggestion was obtained. Deep learning techniques such as RNN, LSTM were applied to code generation model [8] [9] [11] to achieve a higher prediction accuracy. The work in [11] confirmed that LSTM significantly outperforms other models for doing token-level code suggestion. Given large amount of unstructured code, deep language models such as LSTM or its variants are the state-of-art solutions to the problem of code completion. All works described above are trying to solve the general code completion problem in which every token of code should be predicted and completed based on the context in a fixed or changeable length. There are also a lot of works paying attention to the API completion problem. Common sequences of API calls were captured with per-object n-grams in [12]. In [13], API usages was trained on graphs. Naive-Bayes was integrated into n-gram model to suggest API patterns. The migrations of API are studied in [16]. The completion of API full qualified name is studied in [17].

Models for code synthesis. Another important research field to use language models is code synthesis. In recent years, translating text description into source code have achieved a great success. Seq2Seq [18], Seq2Tree [19] and Tree2Tree [20] models are proposed to handle the problem of code synthesis. The models for code synthesis are all based on the framework of neural machine translation. There are two modules named encoding module and decoding module in the framework of neural machine translation models. The encoding module encodes source sentences (trees) into fixed size vector. The decoding module decodes the fixed size vector generated by encoding module into sequences or trees. As discussed in [21], intuitively, Seq2Tree takes grammar of code into consideration but is still based on the framework of Seq2Seq. Although there are big differences between neural code translation and code completion, the aim of the decoding module in code synthesis is same as that of the language model. Both the aims are to generate the suitable code. So it is fair to compare our model with the decoding module in code synthesis model. Traditional decoding module of neural machine translation model uses standard LSTM directly, so there is no need to compare. But the recently proposed tree decoding module in Seq2Tree or Tree2Tree models deserves to be investigated for the code completion task. The decoding module in Tree2Tree model in most recent work [22] are extracted and compared with our REP model. There exists repetitiveness in the synthesized code. Our REP model could be taken as the decoding module for all code synthesis models directly. We will further investigate the performance of taking REP model as the decoding module in code synthesis tasks in the future work. On top of general code synthesis problems, API synthesis is also studied in [3], [4]. In the research fields of natural language processing, text summarization [23] is an important problem which is related to synthesis. One application of text summarization is to automatically generate the title of an article based on the content of that article. In source code processing, there is a similar problem: how to generate the name of a function according to the content of that function. This problem of function name generation has been addressed by extreme summarization of source code based on deep neural attention network in [24].

Models for code classification. For the problem of code classification or the problem of identifying code similarity, TreeNN [25], [26], TreeCNN [27], EQNET [28] or GNN (Graph Neural Network) [29] have been proposed. There exists huge differences between code classification problems and the code generation problems.

Models for robustness. To make the language model more robust, instead of improving the model structure directly, another research direction is to use different sampling schedules [30] or to generate adversarial examples [31] based on the training data to improve the robustness of the model.

Models for capturing token repetition of code. By comparing the related works mentioned all above, as far as we know, the REP model is the first deep neural model based on graph to capture the general token repetition of source code to help improve the solution to the problem of token-level code completion. The newly proposed model is also the first model to address problems of learning usage patterns of variables, templates and cloned code when doing code completion.

III. PROBLEM FORMULATION

Traditional language model processes tokens (words) one by one. The processing is based on a linear chain to handle each token in a token sequence. The data (token sequence) can be converted to the directed acyclic graph (DAG). Each node in the graph has only one incoming edge from its corresponding previous node. Such edge indicates the order of occurrence of tokens in a token sequence and we give that kind of edge a name $CommonEdge$. For example, if $token_n$ has one incoming edge from $token_{n-1}$, then $token_n$ should occur instantly after $token_{n-1}$. Then any token sequence could be represented by a simple graph. Follow this idea, to represent the repetition of tokens, we add an edge named $RepetitionEdge$ between the repeated two tokens. For example, in a token sequence, if $token_m$ is same as $token_n$ where $m < n$, the special designed directed edge $RepetitionEdge$ is added from $token_m$ to $token_n$. To see whether a token $token_n$ is repeated or not in a token sequence is just to see whether there is an edge linked to token $token_n$ from some previously existed token. Then, the problem of learning the general token repetition of source code reduces to the problem of learning the edge connection in a graph. The problem of code generation (code completion) reduces to the graph generation problem. In traditional language modelling, the edges between tokens do not need to be explicitly modeled. In our setting, the patterns of edge connections between tokens need to be explicitly modeled and learned. An example is shown in Figure 1. The circle in Figure 1 represents the token. The edges marked as solid arrow ($CommonEdge$) show the order in which tokens are processed in a token sequence. The repeated tokens are connected by dashed arrow ($RepetitionEdge$). The REP model is still based on the framework of language model and can be taken as a complement to the language model. Tokens in a token sequence are predicted one by one. For each token, REP model additionally judge whether the currently predicted token should be the repetition of some previously existed token. As the language model may predict wrong content (wrong variable names or wrong templates), REP model could help correct the prediction through mixing the patterns of code repetitiveness together. In this setting, the REP model is designed to predict correctly the incoming $RepetitionEdge$ of each token.

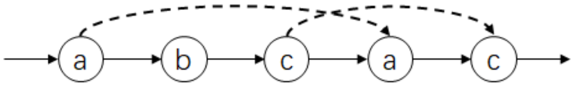

Fig. 1: Graph for Code and its Token Repetition

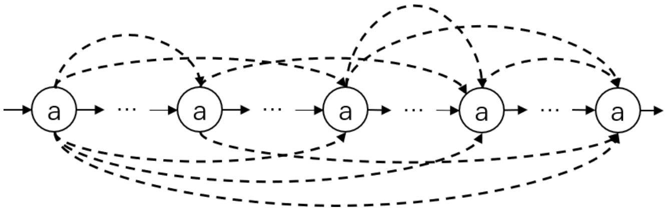

Fig. 2: Complex Graph for Code and its Token Repetition

Note that, one token may have more than one incoming edge if there exist two or more previous tokens with same content as current token in a token sequence. This complex situation is shown in Figure 2. Among all incoming edges ($RepetitionEdge$), if one edge could be predicted correctly, the prediction about the token repetition is right. So it is absolutely unnecessary to predict correctly all edges of kind $RepetitionEdge$. Learning to predict correctly one among all edges of kind $RepetitionEdge$ is enough. To make the learning procedure easier, we could retain and learn only one incoming $RepetitionEdge$. In fact, considering only one of the all possible edges of $RepetitionEdge$ not only meets the needs about deciding the token repetition but also makes the whole problem easier. Therefore, only the nearest two repeated tokens are connected by an edge ($RepetitionEdge$). Formally, for nth token $token_n$, only the kth token $token_k$ connects to $token_n$ where k is determined by

$$k = \max\{\ i \ \mid\ token_i == token_n, i < n\ \} \tag{1}$$

The simplified graph corresponding to the graph in Figure 2 is shown in Figure 3. In Figure 3, only the edges connecting between the nearest two repeated tokens are retained. By removing unnecessary edges, now, every node has at most one incoming $RepetitionEdge$ making the problem concise. When predicting next token, the whole problem of learning token repetition of source code is further divided into two sub-problems: 1. deciding whether there is a $RepetitionEdge$ connected to next token; 2. if it is determined that there must be a $RepetitionEdge$ connected to the next token, deciding which token is the source token of that $RepetitionEdge$ (the next token should be same as the source token, in another word, the source token is the token to be repeated). In addition to the basic language model, learning the repetitiveness of source code (learning extra edge connections between tokens) can extract features of token repetition of source code to recommend code. The ability to learn token repetition can help us identify the usage patterns of variables, templates or cloned code. Details will be described in next section.

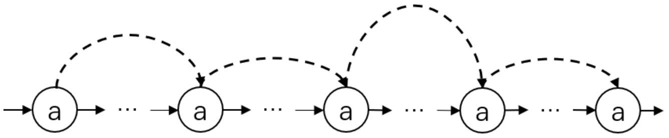

Fig. 3: Simplified Graph for Code and its Token Repetition

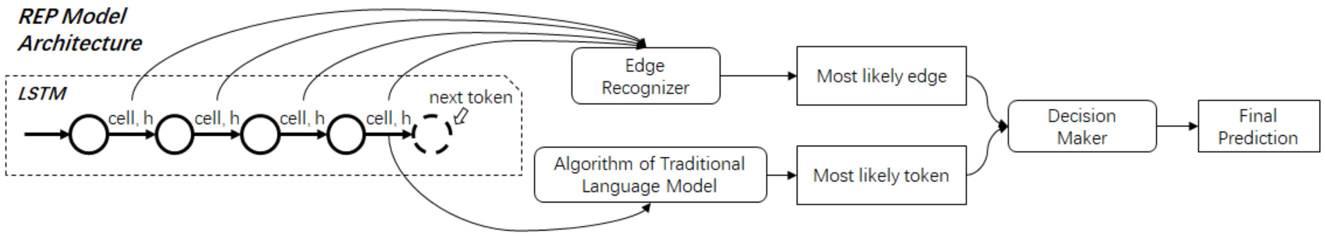

Fig. 4: Overall Architecture

IV. Proposed Method

Given the token sequences parsed from source code, our goal is to learn the general repetitiveness of tokens of source code to improve the performance of existing state-of-art language model. To accomplish this task, we design a novel REP model to learn the repetitiveness of source code tokens. Figure 4 demonstrates the overall architecture of our model. The basic part of REP model is the LSTM model. The LSTM model in REP generates the hidden feature vector (cell, h) for each token. When predicting the edge connection for $token_n$, the Edge Recognizer takes the hidden feature vector (cell, h) of $token_n$ and the hidden feature vector (cell, h) of each previously existed token to compute the probability for each possible incoming edge of $token_n$. The higher the probability, the more likely the edge should be added to the graph. For $token_n$, any edge of kind $RepetitionEdge$ connecting from one of the previous tokens to $token_n$ is the candidate incoming $RepetitionEdge$ of $token_n$. All possible candidate incoming edges ($RepetitionEdge$) of $token_n$ consist of edges of kind $RepetitionEdge$ connecting from all the previous tokens to $token_n$. For example, for $token_3$, all candidate edges consists of two edges: 1. $RepetitionEdge$ connecting from $token_1$ to $token_3$; 2. $RepetitionEdge$ connecting from $token_2$ to $token_3$. The number of candidate edges ($RepetitionEdge$) is $n-1$ for $token_n$. The Decision Maker is aimed at deciding whether the most likely edge predicted by Edge Recognizer should be really added or not. If the Decision Maker decides that there should be no edges ($RepetitionEdge$) connecting to the token being predicted, in this situation, the token predicted by the algorithm of the traditional language model will be taken as the final prediction result. If the Decision Maker decides that the next code token should be the one previously existed, in this situation, the source node (token) of the most likely edge will be thought to repeat at the position currently being predicted. So the token of the source of the recognized most likely edge will be taken as the final prediction result. To learn the general repetitiveness of source code tokens, when predicting a token, we need to take all previously existed tokens into consideration to decide whether or not one of the previously existed tokens should be repeated. The task becomes more and more challenging as the quantity of already predicted tokens becomes larger and larger. Traditional methods such as n-gram can only take limited quantity of tokens into consideration. There is little chance for those methods to discover the repetition of two tokens due to large amount of other tokens between them. Deep learning methods scale well to high dimensional domains and have strong representational power. With the introducing of the deep neural networks, learning the general repetitiveness over a long token sequence becomes possible. In this subsection, we introduce our REP model which is based on the deep neural network.

A. *LSTM in* REP *Model*

The REP model is based on the LSTM model. LSTM model is applied to generate the state ($cell$ and h) for every token in a sequence. Formally, the state generated for predicting $token_n$ by LSTM is referred to as $state_n$ ($cell_n$ and h_n). In rest of this section, the symbol $state_i$ ($cell_i$, h_i) refers to the state generated for predicting ith token $token_i$ by the LSTM. Traditional language model based on LSTM model uses h_n to compute the probability distribution of all candidate tokens to select the most likely token for $token_n$. The LSTM and its usage in traditional language model are omitted in this section. Please refer to [32] for the details of the algorithm of the traditional language model about how to use h_n which is generated by LSTM to compute the most likely token when predicting $token_n$. When predicting the edge connection for $token_n$, h_n, h_{n-1} .. h_1 are used for computing the probability distribution of all candidate incoming edges of $token_n$. Details are shown in the following section.

B. *Edge Recognizer in* REP *Model*

Edge Recognizer is responsible for computing the probability distribution of all candidate incoming edges. Formal definition is as follows. The edge connecting from $token_m$ to $token_n$ where $m < n$ is referred to as $edge_{(m,n)}$. In rest of this paper, $edge_{(x,y)}$ refers to the $RepetitionEdge$ connecting from xth token $token_x$ to yth token $token_y$. The probability of $edge_{(m,n)}$ is computed by:

$$P(edge_{(m,n)}) = \frac{h_m^T \ W \ h_n}{Z} \tag{2}$$

In Equation 2, Z is the normalization factor computed by:

$$Z = \sum_{m=1}^{n-1} h_m^T \ W \ h_n \tag{3}$$

The h_m is in the $state_m$ ($cell_m$, h_m) generated for mth token by LSTM. The h_n is in the $state_n$ ($cell_n$, h_n) generated for nth token by LSTM. In rest of this section, h_i is in the $state_i$ ($cell_i$, h_i) generated for ith token by LSTM. In Equation 2 and

3, W is the model parameter, h_m^T is the transposition of h_m. The training and predicting are all based on the probability of each candidate $RepetitionEdge$ computed by Equation 2.

Training of Edge Recognizer. Remember that, in the section of Problem Formulation, each code will be converted into a token sequence, if $token_n$ is the repetition of some previously existed token, there must be an incoming edge of kind $RepetitionEdge$ connecting to $token_n$. The source of that edge is the nearest previous token which is same as $token_n$. For $token_n$, we use the symbol: $near_n$ to refer to the position of that nearest token which is same as $token_n$. In another word, what we mean is that the nearest token (same as $token_n$) is the $near_n$th token in the token sequence ($token_{near_n}$). According to the position of the source and the target, the incoming $RepetitionEdge$ connecting from $near_n$th token to nth token is referred to as $edge_{(near_n,n)}$. For $token_n$, we define a symbol $e_n \in \{0,1\}$ which indicates whether there is an incoming $RepetitionEdge$ connecting to $token_n$. If e_n is 1, this indicates that there is an incoming $RepetitionEdge$ for $token_n$. If e_n is 0, this indicates that there is no incoming $RepetitionEdge$ for $token_n$. If there is an incoming $RepetitionEdge$ for $token_n$ in training examples, the source (token) of that incoming $RepetitionEdge$ is referred to as $token_{near_n}$ which is the $near_n$th token in the token sequence. The symbol $near_n$ refers to the position of the previously existed token which is nearest to $token_n$ and is same as $token_n$. To learn the edge connections in training examples, we follow the framework of Maximum Likelihood Estimation approaches: the probabilities of actually existed edges of kind $RepetitionEdge$ in training examples should be as high as possible. Thus, the probability of the incoming $RepetitionEdge$: $edge_{(near_n,n)}$ (if there is one) for $token_n$ in training examples should be as high as possible. This is equivalent to minimize the following loss function. If there is no incoming $RepetitionEdge$ for $token_n$, then $near_n$, $edge_{(near_n,n)}$ and P($edge_{(near_n,n)}$) will be default meaningless values. This form avoids the using of condition judgment such as if-else to judge whether the incoming edge is existed or not and takes advantage of the high performance of GPU. With the probability of each edge, the loss function could be defined as (assuming that the quantity of code nodes in a data set is N):

$$\mathcal{L}_{ER} = \sum_{n=1}^{N} -log(P(edge_{(near_n,n)})) * e_n \tag{4}$$

The symbol e_n indicates whether $token_n$ has an incoming $RepetitionEdge$. If there is no incoming $RepetitionEdge$, e_n is 0 and the final loss will exclude the loss for non-existent edges. The training objective of Edge Recognizer is to minimize the loss function 4.

Usage of Edge Recognizer in Prediction Phase. The edges with highest probabilities computed by Equation 2 are most likely to be added to the graph. In another word, the source of the edge with high probability has a high chance to be repeated. When predicting $token_n$, for the candidate incoming $RepetitionEdge$ with the kth largest probability, we use the symbol: src_k to refer to the position of the source of that kth most likely edge. In another word, the source of the kth most likely $RepetitionEdge$ when predicting $token_n$ is the src_kth token ($token_{src_k}$). If $k = 1$, we can offer another formal definition of src_1 when predicting $token_n$:

$$src_1 = \arg\max_i \quad P(edge_{(i,n)}) \tag{5}$$

When predicting $token_n$, if Decision Maker (described in the following subsection) decides that the $token_n$ should be the repetition of some previously existed token and top-k candidates are needed, $token_{src_1}$ (the source token of the edge with highest probability), $token_{src_2}$ (the source token of the edge with the second highest probability), ... and $token_{src_k}$ (the source token of the edge with kth highest probability) will be taken as the final recommendation. The top-k accuracy is computed by judging whether the desired $token_n$ exists in the candidates: $token_{src_1}$, $token_{src_2}$, ... and $token_{src_k}$.

C. *Decision Maker in REP Model*

The task of Decision Maker is to decide whether the edge with the highest probability computed by Edge Recognizer should be really added to the graph or not. When predicting $token_n$, Decision Maker is to decide whether there should be an edge of kind $RepetitionEdge$ connecting to $token_n$. In the phase of predicting, the available information about $RepetitionEdge$ is the probability of each candidate $RepetitionEdge$ computed by Edge Recognizer for $token_n$. The probability that $token_n$ has an incoming $RepetitionEdge$ ($token_n$ is the repetition of some previously existed token) is computed by:

$$P(token_n \ is \ repeated) = \frac{h_{src_1}^T \ V_1 \ h_n}{h_{src_1}^T \ V_1 \ h_n + h_{src_1}^T \ V_2 \ h_n} \tag{6}$$

In the above equation, h_{src_1} is the h in $state$ ($cell$, h) generated for the token at the position src_1 (src_1th token) and $h_{src_1}^T$ is the transposition of h_{src_1}. The src_1 is defined in Equation 5. V_1 and V_2 are model parameters.

Training of Decision Maker. If there is an incoming edge in training example for $token_n$, $P(token_n \ is \ repeated)$ defined in Equation 6 should be maximized. Otherwise, the value: $1 - P(token_n \ is \ repeated)$ should be maximized. This corresponds to minimize the following loss. The loss function of Decision Maker is defined as:

$$\begin{aligned} \mathcal{L}_{DM} = \sum_{n=1}^{N} (&-log(P(token_n \ is \ repeated) * e_n \\ &-log(1 - P(token_n \ is \ repeated)) * (1 - e_n))) \end{aligned} \tag{7}$$

The training objective of Decision Maker is to minimize the loss function 7. The final loss $\mathcal{L} = \mathcal{L}_{ER} + \mathcal{L}_{DM}$. To minimize the final loss $\mathcal{L}$ is equivalent to minimize $\mathcal{L}_{ER}$ and $\mathcal{L}_{DM}$ separately.

Usage of Decision Maker in Prediction Phase. When predicting $token_n$, if P($token_n \ is \ repeated$) is greater or equal to 0.5, the recommendation result generated by Edge

Recognizer (described in the previous subsection) should be taken as the final result. Otherwise, if P($token_n$ $is\ repeated$) is less than 0.5, the prediction result generated by the algorithm [32] of the traditional language model will be taken as the final prediction result.

D. Advanced REP Model

In previous subsections, there is only one LSTM model described in REP. To improve the expressiveness of the REP model, we could use two LSTM models, one for computing probabilities of edges in REP, one for computing the prediction result of the standard language model. In another word, the LSTM used for computing the prediction result of standard language model is the LSTM isolated from the LSTM model described in REP. Also, the embeddings of tokens involved in those two LSTM models could be isolated. In conclusion, we use one LSTM model as the standard language model and use another LSTM model to compute the probabilities of edges in REP. By doing so, the parameters have doubled in size and the performance of the REP model could be further improved. Actually, the REP model used in experiments is the advanced version just described in this subsection.

V. IMPLEMENTATION

Source code [33] which contains all data sets and all implementations of all models mentioned in experiments has been public. The implementation of the model is based on the deep learning platform TensorFlow. Apart from the parameters of the LSTM used in REP model, the other parameters of REP model are W in equation 2 and V_1, V_2 in equation 6. Adam optimizer in TensorFlow is used to automatically decide learning rate and momentum in training phase. The gradient is clipped by global norm. Parameters about clipping norm are set to default values offered by TensorFlow. The representation size (alias as embedding size or feature size) for one token is 128. Once the embedding size for one token is decided, sizes of all other parameters which participate in the calculation with the embedding of tokens can be decided successively. All models keep training until the accuracy on validation set does not exceed the optimal value for 50 epochs. All the training examples are trained one by one. The models are running on the computer with the setting: Windows 10 64 bit OS, Intel i7-6850k CPU, 32G memory and one Geforce GTX 1080 Ti GPU. Every function in source code will be extracted and tested. The code completion system will start at the beginning of a function to try to complete each token of the function one by one. The accuracy is the average of the prediction accuracy of each token of each function in test set. To generate tokens for the source code, some works [5] parse the code into lexical units through splitting the code by white space or other predefined separators such as $+$, (, $:$ or ;. The implementation of such parser may vary greatly. Different separators in use lead to different token sequence of source code. Thus, to make the token generation for code unified, the token sequence is generated in the following steps. The code is parsed into the abstract syntax tree (AST) through Eclipse JDT at first. Then, the AST is traversed in pre-order. The content of each encountered node is pushed back onto a sequence. Now, the token sequence has been generated. Token repetition learning is also based on the token sequence generated in this way. Generating token sequence based on AST could also make it fair to compare the sequential model such as LSTM with tree models such as Tree2Tree model as the contents to be predicted are same in this setting.

VI. EXPERIMENT

Without loss of generality, the most widely used programming language: Java is chosen to conduct experiments. Four data sets are provided for evaluations. For each data set, the source code in that data set is divided into training set, validation set and test set in the proportions 60%, 15%, 25%. Follow the one billion word benchmark [34], similarly, the 0.15% least frequently occurred code tokens in our training set, all unseen tokens in validation set and test set are marked as UNK.

Data Sets. Famous open-source projects are used in experiments. As there are small functions which contain only one or two statements in those projects, we do some filtering to the code. The filtering steps are as follows: 1. Give each Java file a score (to get the score, divide the number of all tokens in all functions in the java file by the number of all functions in that java file); 2. Sort Java files according to the score from large to small and extract the first 85% Java files to blend into a data set. Details are shown in Table I. The fourth column headed by *Vocabulary* in Table I means the total number of unique tokens on the data set. The symbol DS refers to data set. For example, the DS1 refers to data set 1. The original size of Apache Lucene is 98.8MB and is too huge, so only the core module and analysis-common module are extracted into the data set.

TABLE I: Data Sets

	From Project	Size	Vocabulary
DS1	Log4J	1.76MB	6455
DS2	Maven	3.25MB	10516
DS3	FindBugs (GitHub version)	8.54MB	27573
DS4	Lucene (core & analysis-common)	15.6MB	55244

Research Questions. To demonstrate the ability of the proposed model, one research question is answered:

RQ1: Could REP model achieves better prediction accuracy than other state-of-art methods under all the data sets?

To evaluate the performance of our model, the state-of-art baselines: LSTM and tree decoding module in Tree2Tree [22] are compared with the REP model. The tree decoding module in other Tree2Tree models [35] is just LSTM model with silghtly changes (one LSTM for decoding content of node, one LSTM for decoding the tree structure of node) and is irrelevant to this research because we only care about predicting the content of node in this research. Table II shows the prediction accuracy of different models evaluated on the test set of each data set. The top-k accuracy (value is in percentage, % is omitted to save the space) is computed for evaluating

TABLE II: Evaluation Result

	LSTM						Tree2Tree						REP					
	top1	top3	top6	top10	mrr	enpy	top1	top3	top6	top10	mrr	enpy	top1	top3	top6	top10	mrr	enpy
DS1	45.4	59.3	63.7	66.3	0.53	6.7	34.7	54.3	61.5	64.7	0.45	5.9	**48.6**	63.0	68.1	70.9	0.57	**2.8**
DS2	48.0	60.7	65.2	67.7	0.55	6.8	36.7	54.4	59.8	63.0	0.46	5.8	**52.6**	66.3	71.2	73.8	0.60	**3.3**
DS3	34.5	49.2	55.2	58.6	0.43	7.7	31.8	50.0	56.4	59.2	0.39	6.9	**44.7**	59.9	66.3	69.9	0.53	**5.8**
DS4	48.9	63.7	69.8	73.3	0.57	3.6	40.0	58.0	65.1	69.3	0.50	4.0	**53.9**	69.1	75.3	78.6	0.62	**1.7**

the model performance. When predicting next node, we rank all candidate tokens according to probabilities (computed by model) from large to small. The token with higher probability has smaller rank. The token with the highest probability has rank 1. The value in column headed with *mrr* is the average of the reciprocal of rank of the token. This metric indicts the overall prediction performance of the model. The larger the *mrr*, the better the performance of the model. The *enpy* headed column shows the entropy (log value of the perplexity) of the model. The smaller the entropy, the better is the model.

As can be seen from the data, the REP model outperforms all other state-of-art models in all 4 data sets. Especially for the top-1 accuracy, REP model achieves averagely 17.2% improvement compared to LSTM, achieves averagely 39.7% improvement compared to Tree2Tree. From the result, we can conclude that capturing the token repetition of source code can improve existing LSTM model which has solved the problem of gradient vanishing and gradient exploding to capture the long term memory. Source code is diverse. After randomly checking files in test set and training set, we have discovered that there is little chance for the exact same code appears in both the training set and test set. As argued in [30], if some sub-sequences or patterns in the context are unseen in training data, the discrepancy between training and inference could cause the system fail to predict the right result. This problem is named as $exponential\ bias$ and REP model could solve that problem to some extent.

The tree decoding module in Tree2Tree [22] performs worse than LSTM. From this experiment, tree decoding module in Tree2Tree is strongly dependent on the attention mechanism used in its encoding module. Without the cooperation from the encoding module, the performance of a single decoding module is worse than standard LSTM. The reason has been carefully analyzed. In Tree2Tree model [22], trees need to be converted into binary trees. Converting a general tree to a binary tree needs to add more nodes makes the prediction task more difficult. The decoding module decodes the binary tree in the way that the left child node and right child node of one node will be predicted at the same time. This decoding procedure indicates that the right child is predicted without the information of the left child and vice versa. In the mean while, LSTM model predicts one by one meaning that the right child is predicted with the information of the left child (according to pre-order traversal of tree). The less use of the context information causes the lower prediction accuracy of tree decoding module in Tree2Tree [22]. On the other hand, the less use of the context could reduce the impact of the unseen data in context. When encountering a large amount of unseen data, Tree2Tree could gain good generalization ability. That is why Tree2Tree performs better than LSTM in entropy on the first three data sets. Even so, Tree2Tree still performs worse than REP in entropy. As can be seen from the entropy (log value of the perplexity), REP achieves nearly half the entropy compared to that of other models. This experimental result is dramatically encouraging. Thus, by taking all conditions into account, REP outperforms other models.

RQ2: In which scenarios can REP model achieve better results than other models and what features of scenarios bring REP model to the better performance?

In the investigation of the prediction result of the experiment, we discover that REP model is good at predicting unseen data especially for unseen variable names or unseen type names. In our setting, unseen token in validation set or test set will be replaced with UNK. Both LSTM model and Tree2Tree model perform badly in distinguishing between UNK and other tokens when the number of tokens is huge. In REP model, by recognizing the hidden relationships of token repetition, another point of view could be offered to us to decide which is the most likely token for the next. If the model can confirm that the next token should be the repetition of some previously existed token, we could take the previously existed token as the final prediction result no matter the previously existed token is UNK or other tokens. Note that, for a token sequence of which the length is often less than 1000, the number of the previously existed tokens is at most 1000. In this setting, the REP model only needs to distinguish between at most 1000 tokens to decide the token repetition. In the meanwhile, for large projects, the number of total tokens is often greater than 6000, which means that the standard LSTM-based language model needs to distinguish between at least 6000 tokens to decide which is the most likely token for the next. Obviously, the task of recognizing token repetition is much easier than the task of the standard LSTM-based language model. That is the one factor why REP model could perform better than standard LSTM-based language model.

Limitation and Future Work. In experiments, only four Java projects are used, more projects could make the results of experiments more solid. In all projects of different sizes, the proposed model has achieved better results than all other models, which can prove the effectiveness of the model to a certain extent. The proposed method is not limited to the Java language and can be extended to other languages such as Python and C++. The extra work needed to do is to design Python parser or C++ parser to parse the code into the corresponding token sequence. The proposed model could be applied to the generated token sequence. In the future, the performance of the proposed model could be further investigated on different languages such as Python or C++. If the code corpus is large, there would be a lot of tokens. The existence of a large number of tokens can cause trouble to apply this technique to industrial scenarios. The techniques which are designed to minimize the number of tokens could be applied to further reduce the total tokens to improve the model performance.

VII. CONCLUSION

In this paper, a novel REP model is proposed to capture the general token repetition of source code to improve the prediction accuracy of standard language model. The experimental results on huge data sets confirm that capturing the general token repetition of source code by REP model successfully improves current methods and makes a step forward for the problem of code completion.

REFERENCES

[1] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *Computer Science*, 2014.

[2] D. Pawade, A. Sakhapara, M. Jain, N. Jain, and K. Gada, "Story scrambler–automatic text generation using word level rnn-lstm," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 10, no. 6, pp. 44–53, 2018.

[3] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2api: synthesizing api code usage templates from english texts with statistical translation," in *ACM Sigsoft International Symposium on Foundations of Software Engineering*, 2016, pp. 1013–1017.

[4] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334

[5] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 837–847. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227135

[6] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[7] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *The ACM Sigsoft International Symposium*, 2014, pp. 269–280.

[8] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Ieee/acm Working Conference on Mining Software Repositories*, 2015, pp. 334–345.

[9] H. K. Dam, T. Tran, and T. T. M. Pham, "A deep language model for software code," in *FSE 2016: Proceedings of the Foundations Software Engineering International Symposium*. [The Conference], 2016, pp. 1–4.

[10] V. Raychev, P. Bielik, and M. T. Vechev, "Probabilistic model for code with decision trees," in *OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 731–747. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984041

[11] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[12] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 44. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[13] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 858–868. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.336

[14] M. Allamanis and C. A. Sutton, "Mining source code repositories at massive scale using language modeling," in *MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 207–216. [Online]. Available: http://dx.doi.org/10.1109/MSR.2013.6624029

[15] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause, "Learning programs from noisy data," in *POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 761–774. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837671

[16] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *IEEE/ACM International Conference on Software Engineering*, 2017.

[17] H. Phan, H. Nguyen, N. Tran, L. Truong, A. Nguyen, and T. Nguyen, "Statistical learning of api fully qualified names in code snippets of online forums," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 632–642.

[18] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[19] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[20] M. Drissi, O. Watkins, A. Khant, V. Ojha, P. Sandoval, R. Segev, E. Weiner, and R. Keller, "Program language translation using a grammar-driven tree-to-tree model," *arXiv preprint arXiv:1807.01784*, 2018.

[21] J. Sedoc, D. Foster, and L. Ungar, "Neural tree transducers for tree to tree learning," 2018.

[22] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[23] J.-g. Yao, X. Wan, and J. Xiao, "Recent advances in document summarization," *Knowledge and Information Systems*, vol. 53, no. 2, pp. 297–336, 2017.

[24] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.

[25] R. Socher, B. Huval, C. D. Manning, and A. Y. Ng, "Semantic compositionality through recursive matrix-vector spaces," in *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*. Association for Computational Linguistics, 2012, pp. 1201–1211.

[26] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.

[27] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1287–1293.

[28] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. Sutton, "Learning continuous semantic representations of symbolic expressions," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 80–88.

[29] X. Xu, L. Chang, F. Qian, H. Yin, S. Le, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," 2017.

[30] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," 2015.

[31] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient." in *AAAI*, 2017, pp. 2852–2858.

[32] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[33] "The source code of models in the experiments and all data sets," https://www.dropbox.com/s/28p8j44zdc78ob4/REP.zip.

[34] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, "One billion word benchmark for measuring progress in statistical language modeling," *arXiv preprint arXiv:1312.3005*, 2013.

[35] D. Alvarez-Melis and T. S. Jaakkola, "Tree-structured decoding with doubly-recurrent neural networks," 2016.