SLAMPA: Recommending Code Snippets with Statistical Language Model

Shufan Zhou, Hao Zhong, Beijun Shen*
School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
{sfzhou567, zhonghao, bjshen}@sjtu.edu.cn

***Abstract*—Programming is typically a difficult and repetitive task. Programmers will encounter endless problems during programming, and they often need to write similar code over and over again. Over the years, many tools have been proposed to support programming, but none of the existing works are able to precisely recommend relevant and compilable code snippets during the process of programming.**

To address this challenge, we propose SLAMPA, a novel tool which takes advantage of statistical language model and clone detection techniques to recommend code snippets during programming. Given a piece of partially written code, SLAMPA first infers its intention using a neural language model. Then it retrieves code snippets from codebase with the support of an efficient clone detection technology Hybrid-CD we proposed. Finally, it recommends the most similar code snippets to programmers.

Our experiments demonstrate that the Hybrid-CD we proposed is able to precisely detect similar code snippets and it outperforms previous techniques. Our experiments further indicate that the snippets recommended by SLAMPA catch the demands of programmers and SLAMPA is capable of finding potential code reuse opportunities during programming.

***Index Terms*—code snippets recommendation, code reuse, statistical language model, clone detection**

I. Introduction

In software development activities, programmers may encounter endless problems and they may turn to source code examples for help. Source code examples are critical for understanding concepts, applying fixes, improving performance, and extending software functionalities [1][2]. With the development of code repositories such as Github[1], SourceForge[2], millions of high-quality code snippets become available.

Unfortunately, finding appropriate code examples to cope with the problems encountered during programming is still a serious challenge for programmers. Although some existing works have investigated using free-form query (i.e., a query written in natural language, or a list of keywords) to search relevant code snippets [3][4], it is hard for programmers to precisely summarize what have been done in the partially written code into one query. For example, giving a partially written code shown in Figure 1, it is hard to accurately depict this piece of code using natural language. Some publicly available code search engines take as input the entire code snippet [5][6], but these code search engines are only able to identify identical code fragments [7]. Furthermore, since it is impossible for programmers to be aware of all the relevant code snippets in code repositories, programmers may miss the potential code reuse opportunities of the partially written code and reinvent wheels during programming [8].

```
public ImageData getJPEGDiagram() {
  Shell shell = new Shell();
  GraphicalViewer viewer=new ScrollingGraphicalViewer();
  viewer.createControl(shell);
  LayerManager lm = (LayerManager) viewer.
        getEditPartRegistry().get(LayerManager.ID);
  IFigure fig = lm.getLayer(LayerConstants.
        PRINTABLE_LAYERS);
  ...
}
```

Fig. 1. A piece of partially written code (id=16163062 in BigCloneBench)

To handle these challenges, we propose a novel code snippets recommendation tool named SLAMPA (Statistical LAnguage Model based Programming Assistant). The recommended code snippets by SLAMPA can be used for understanding concepts, extending software functionalities, etc. SLAMPA can also detect potential code reuse opportunities and recommend relevant code snippets to increase programmers' productivity.

SLAMPA works by a combination of statistical language model and clone detection techniques. As Hindle et al. reported that programs, as a natural product of human effort, exhibit a good level of repetition [9]. SLAMPA captures the repetition of programs by using a deep neural network-based statistical language model, which detects long-range functional features in a context-dependent way. With the language model, SLAMPA can analyze the intention of a piece of partially written code and further infers the tokens which were possibly be used later. SLAMPA then implicitly expands the given partially written code with the inferred tokens and uses Hybrid-CD (Hybrid-Clone Detector), an efficient clone detection technology we proposed, to retrieve similar code snippets. Finally, the most relevant code snippets are recommended to programmers.

We evaluate the effectiveness of Hybrid-CD and SLAMPA on BigCloneBench [10], a large scale inter-project repository with manually labeled clone pairs. Our evaluation results demonstrate that Hybrid-CD outperforms previous techniques. Hybrid-CD is able to detect almost all the T1-ST3 clones (94% - 100%) and more than half of the MT3

*Corresponding author.
[1]https://github.com
[2]https://sourceforge.net

clones (51%). Our experiments further show that SLAMPA can indeed capture the demands of programmers via the statistical language model and find code reuse opportunities. For 68.2% of the queries, the relevant snippets can be found within the top 10 recommended results.

In summary, the contributions of this paper lie in:

- We propose and implement Hybrid-CD, a novel clone detection technology leveraging deep learning. Hybrid-CD combines automatically extracted high-level features (by a deep neural network) and handcrafted low-level features. Hybrid-CD can effectively detect clones in real-world repositories.
- We propose SLAMPA, a novel code recommendation tool that makes it possible to automatically and accurately recommend code snippets during programming. To the best of our knowledge, we are the first to propose a combination of language model and code clone detection to recommend code snippets.
- We implement a prototype of SLAMPA and evaluate it on real-world repositories. Our experimental results show that SLAMPA is of potential use in software development.

II. Background

Our approach combines recent advanced techniques from programming language processing and deep learning. We will introduce the background of these techniques in this section.

A. Code Clones

There are four main types of code clones: Type-1 (T1), Type-2 (T2), Type-3 (T3), and Type-4 (T4) [11]. T1 clones are the exact copies of each other except whitespaces, blanks and comments. T2 clones are similar code fragments except for names of variables, types, literals and functions. T3 clones allow extra modifications such as added, or removed statements. T4 clones further include semantically equivalent but syntactically different code fragments. T3 clones are divided into Strongly T3 (ST3), Moderately T3 (MT3) and Weakly T3/T4 (WT3/T4) according to syntax similarities in BigCloneBench [10].

B. Statistical Language Model

Hindle et al. have introduced the naturalness of software in [9]. The naturalness of software shows that source code, just like many other forms of culturally contextualized and stylized natural language expression, tend to be well-structured and repetitive. We can capture the naturalness of software by using statistical language models. A statistical language model learns to estimate the distribution of code from large code corpus and essentially assigns a probability to programs.

Given a token sequence $S = t_1, t_2, ..., t_m$ of a program, the statistical language model estimates the joint probability

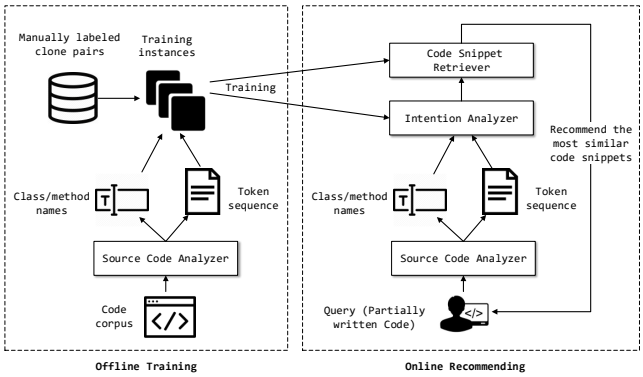

Fig. 2. The overall workflow of SLAMPA

$Pr(S) = Pr(t_1, t_2, ..., t_m)$ as how likely this sequence would occur in a programming language. Formally:

$$Pr(S) = Pr(t_1, t_2, ..., t_m) = \prod_{i=1}^{n} Pr(t_i | t_1, ..., t_{i-1}) \quad (1)$$

where $Pr(t_i|t_1, ..., t_{i-1})$ is the probability of each token in the programming language given its preceding words. With the ability to calculate such a distribution, we will be able to predict with high confidence of what follows the given code.

1) Neural Language Model: The neural language models are language models based on neural networks. Neural language models capture the information under the preceding words with longer distance to predict the following word.

A typical neural language model reads one token from the input sentence at each time step and predicts the next token. At time step j, a neural language model reads the input token t_j, embeds it into word embedding e_j and updates the current hidden state of network according to the previous hidden state h_{j-1} and the current embedding e_j. Finally, the neural language model uses a result predicting function g to predict $Pr(t_{j+1}|t_1, ..., t_j)$ according to the current hidden state h_j. Formally:

$$e_j = input(t_j) \quad (2)$$

$$h_j = f(h_{j-1}, e_j) \quad (3)$$

$$Pr(t_{j+1}|t_1, ..., t_j) = g(h_j) \quad (4)$$

The neural language model repeatedly reads, embeds, and updates until it meets the end of the token sequence.

C. Embedding Techniques

Embedding (also called distributed representation) is a technique for learning vector representations of entities in a continuous space where linguistic contexts of words can be observed [12][4].

We use fastText [13] in this paper. This approach tries to learn representations for character n-grams, and represents words as the sum of the n-gram vectors. The fastText takes into account subword information (morphology), so it alleviates the out-of-vocabulary (OOV) problem in programming

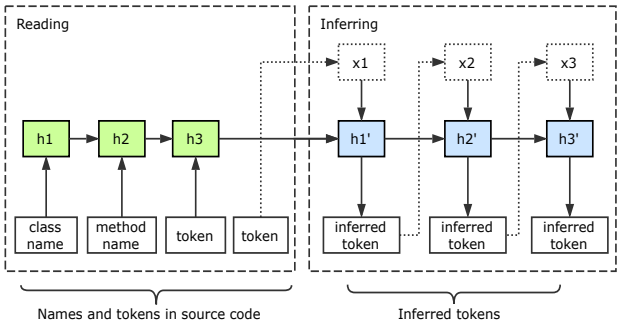

Fig. 3. An illustration of our RNN-based Intention Analyzer, where h and h' are the hidden state of neural network, and x means the input.

language (e.g. $FileReader$ is likely to has similar word embedding with $StreamReader$ in fastText, fastTest can understand the meaning of unseen words to a certain degree).

III. Approach

In this section, we describe SLAMPA, a code snippet recommendation tool based on the statistical language model and the proposed clone detection technology.

Figure 2 shows the overall workflow of SLAMPA. The source code analyzer tokenizes the partially written code into token sequence $\{t_1, t_2, \cdots, t_m\}$. Then the intention analyzer infers a sequence of tokens $\{p_1, p_2, \cdots, p_n\}$ and uses them to expand the given token sequence. Finally, the code snippet retriever takes the expanded token sequence $\{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_n\}$ as input to retrieve similar code snippets, and recommend them to the programmer.

A. Source Code Analyzer

Source code contains a lot of semantic information. We especially focus on API invocations and the corresponding structural information in source code in this work. We have implemented a source code analyzer based on Eclipse JDT compiler [14] to extract both structural information and API invocations from codes at the same time. All the extracted data are called token in the rest of this paper.

1) Token Extraction: For every parameter declaration in abstract syntax trees (ASTs), the code analyzer records its name and type. Then it visits each method declaration in AST and traverses the method body to extract structural information and API invocations. To capture the structural information of codes, the code analyzer add METHOD_BEGIN and METHOD_END around the token sequence of each method. It then records some control-dependency-relevant reserved words including $if, else, for, while, continue, break, try$ and $catch$ to each structural code blocks. Take $while$ for example, the code analyzer will add a WHILE_BEGIN token to the token sequence before traversing the body of the while node and append a WHILE_END token to the token sequence after visiting the node. We call these BEGIN and END tokens as $structural\ bracket$ in the rest of this paper.

The code analyzer applies following rules to extract API invocations: (1) For the constructor invocation new Obj(), we append a token Obj.<init> to the token sequence. (2) For the method invocation o.method(), where o is an instance of Obj, we append a token Obj.method() to the token sequence. (3) For some complicated expression such as o.m1(o.m2(),o.m3()), where o is an instance of Obj, we generate a three tokens Obj.m2(), Obj.m3(), Obj.m1() and append them to the token sequence that we have made. (4) For some sequential API call such as o.m1().m2(), we mark it as token *.m2() if we fail to get the return type of o.m1(), where * serves as a wildcard character and it can match with any other types.

2) Token Filtering: As discussed above that we retain the structural information of source code by adding $structural\ bracket$ around the API invocations. We mainly focus on the API level information during tokenization, but sometimes there is no API invocation in the certain structural code blocks. For example, we might obtain [METHOD_BEGIN, FileWriter.<init>, FOR_BEGIN IF_BEGIN, IF_END, FOR_END, FileWriter.close(), METHOD_END] after tokenizing some code fragment. It is obvious that the empty structural brackets [FOR_BEGIN IF_BEGIN, IF_END, FOR_END] bring no benefit for code snippets recommendation, so we additionally take a process to remove these redundant tokens.

B. Intention Analyzer

We implement our intention analyzer by using a deep neural network called Recurrent Neural Network (RNN) [15]. As shown in Figure 3, when a recommendation is requested, RNN analyzes the names and tokens existed in the partially written code. Then it infers an ordered sequence of tokens which are likely to be used.

1) RNN-based Language Model: We adopt Long Short-Term Memory (LSTM) [16] as the implementation of RNN to model the sequence of tokens. LSTM structure has explicit memory cells for storing information for long periods of time, and it can easily memorize information for an extended number of time steps. Therefore it is more powerful for language modeling. LSTM has shown a great performance for modeling sequential data in recent researches [17][18].

LSTM processes one word at each time step. The hidden states of LSTM are updated as follows:

$$\begin{aligned} f_j &= sigm(W_f e_j + U_f h_{j-1} + b_f) \\ i_j &= sigm(W_i e_j + U_i h_{j-1} + b_i) \\ o_j &= sigm(W_o e_j + U_o h_{j-1} + b_o) \\ c_j &= f_j \cdot c_{j-1} + i_j \cdot tanh(W_c e_j + U_c h_{j-1} + b_c) \\ h_j &= o_j \cdot tanh(c_j) \end{aligned} \tag{5}$$

where e_j represents the input vector at time step j, symbols f, i, o stand for the forget gate, input gate and output gate in LSTM, c_j and h_j mean the hidden states of LSTM at time step j. We adopt softmax function on the state and take it as the probability distribution of the predicted tokens.

Given names and token sequence $\{t_1, t_2, \cdots, t_m\}$ of the partially written code, where m is the number of tokens in source code, LSTM infers a sequence of token $\{p_1, p_2, \cdots, p_n\}$ and appends it to the end of $\{t_1, t_2, \cdots, t_m\}$ to get the new sequence $\{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_n\}$. This new sequence will be further used for code recommendation.

2) Language Model Enhancement: We use focal loss and beam search to enhance our language model.

Focal Loss. In programming language, different tokens appear at different frequencies, and the amount of information they carried is also different. The token prediction is obviously an imbalanced problem. The basic LSTM model does not consider this problem and still uses the traditional balanced cross entropy, which causes the model to easily generate high-frequency, low-information tokens.

We augment the LSTM model by using focal loss. Focal loss proposed by Lin et al. [19] is a new loss function that addresses class imbalance by reshaping the standard cross entropy loss such that it down-weights the loss assigned to well-classified examples. Now the loss function of the LSTM model is:

$$FL(p) = -(1-p)^{\gamma} log(p) \tag{6}$$

where $p \in (0, 1)$ is the model's estimated probability for the class of ground truth, and γ is a hyperparameter. And we set $\gamma = 2$, which is the recommended value in the origin paper.

Beam Search. Using beam search, the intention analyzer can select w best candidates at each time step, where w stands for beam width. Then intention analyzer prunes off the remaining branches and continues selecting the possible tokens that follow on. We set $w = 5$ in this work. This generation procedure repeats n times, where n is the number of tokens to be generated.

3) Language Model Training: We train our language model over two configurations, LSTM-Base and LSTM-FL, where FL is shorthand for focal loss. The difference between these two configurations lies in that LSTM-Base uses the balanced cross entropy as its loss function, while LSTM-FL adopts focal loss. We build our models in TensorFlow [20]. We initialize the hidden states of neural network to zero, and use the final hidden states of the current minibatch as the initial hidden state of the subsequent minibatch in training process. There are 650 units per layer in LSTM and its parameters are initialized uniformly in [-0,05, 0.05]. We also apply 50% dropout on the non-recurrent connections as recommended in [15].

C. Code Snippet Retriever

Given two code snippets s_A, s_B, we believe that the similarity between the class/method names, and the similarity between the token sequence should reflect the similarity between these two code snippets. Thus, finding similar code snippets can be treated as a cloned code detection problem. We propose Hybrid-CD (Hybrid Clone Detection), a brand new clone detection technology, as the code snippet retriever.

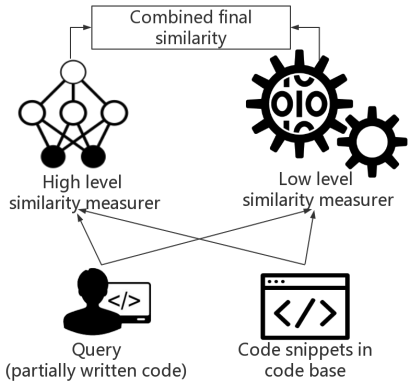

Fig. 4. The workflow of Hybrid-CD

Figure 4 shows the workflow of Hybrid-CD, which consists of two main components: *High-Level Similarity Measurer* and *Low-Level Similarity Measurer*. Hybrid-CD combines the high-level similarity and the low-level similarity to get the final similarity:

$$\begin{aligned} Sim_{final}(s_A, s_B) =& 0.5 * Sim_{high}(s_A, s_B) \\ &+ 0.5 * Sim_{low}(s_A, s_B) \end{aligned} \tag{7}$$

For each query and the corresponding augmented token sequence $\{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_n\}$, we retrieve the most similar code snippets and recommend them to programmers.

1) High-Level Similarity Measurer: Intuitively, when two snippets have similar class/method names and token sequence, they exhibit the similar usage [2]. This component consists of a feature embedding network and a similarity measuring network. The workflow of this component is shown in Figure 5.

The feature embedding network embeds class name, method name as well as the token sequence into a fixed-length vector. We split the names into words according to the capital letters in the names and transform them into embeddings by using fastTest. For tokens, we use bi-directional LSTM (BiLSTM) [21] to embed the entire token sequence into a fixed-size embedding. As shown in Figure 6, the BiLSTM reads the given tokens in both forward and backward directions, and then the final states of two directions are combined by a max pooling layer. Finally, the class/method embeddings and the token embedding are concatenated into a larger embedding called snippet embedding.

The similarity measuring network takes two snippet embeddings as input and calculates the similarity between them. It contains three fully connected layers. The first two layers reconstruct the input features, and the third layer serves as a classification layer to estimate the probabilities of clone or non-clone (1 and 0) for the given embeddings. Then the similarity is calculated as follows:

$$Sim_{high}(s_A, s_B) = \frac{exp(l_1)}{\sum_{i=0}^{1} exp(l_i)} \tag{8}$$

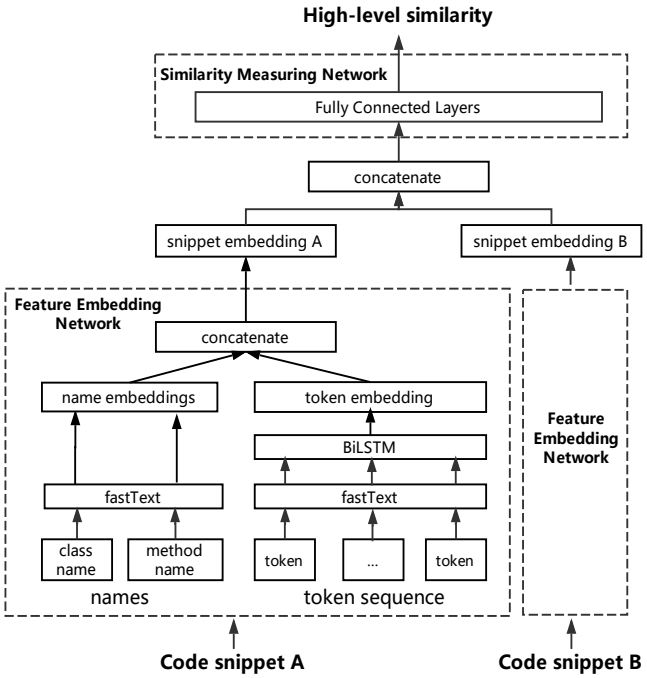

Fig. 5. The workflow of High-Level Similarity Measurer

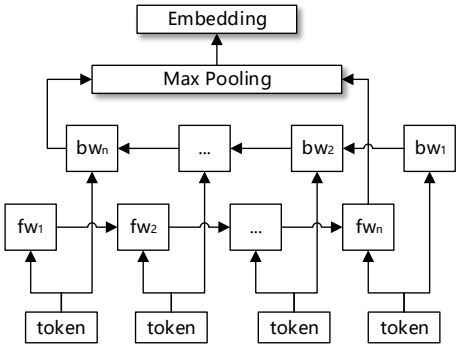

Fig. 6. An illustration of the BiLSTM we used in Feature Embedding Network. fw is shorthand for forward and bw is shorthand for backward.

where $l_i \in \mathbb{R}$ is the value of clone or non-clone predicted by neural netowrk. The high-level similarity is calculated via the softmax value of l_1 (clone).

2) Low-Level Similarity Measurer: The frequencies of tokens contain some information. For the extracted token sequence, we follow [22] to record the frequency of each token and convert the token sequence into token-frequency dictionary D. We then calculate the similarity score as follows:

$$Sim_{low}(s_A, s_B) = 1 - \frac{\sum_x |freq(D_A, x) - freq(D_B, x)|}{\sum_x |freq(D_A, x) + freq(D_B, x)|} \quad x \in tokens(D_A) \cup tokens(D_B) \tag{9}$$

where $freq(D, x)$ is a function which returns the frequency of token x in token-frequency dictionary D. Note that if two sequences have no same tokens, we set the corresponding similarity score as 0.5 by default.

3) Hybrid-CD Model Training: The goal of Hybrid-CD is: given two code snippets (including the class name, method name and the codes), Hybrid-CD should be able to predict a high similarity if these two snippets are clones, and a little similarity otherwise.

At training time, we construct each training instance as a triple $< s_A, s_B, label >$: the necessarily information of two code snippets and the corresponding label (1 or 0, means clone or non-clone). Since these names used in code are close to natural language, we directly use fastTest embeddings pre-trained on natural language corpus[3] for class/method name embedding. We further use fastText[4] on the token sequences to obtain the embedding for each token. The dimension of the pre-trained name embeddings are 300 and we set the dimension of token embeddings to 650. The number of hidden units used in BiLSTM is set as 200.

The similarity measuring network contains three fully connected layers. We separately set the number of hidden units as 200 and 100 for the first two layers. We apply 50% dropout on both of these two fully connected layers. The third layer serves as a classification layer so we set its hidden units as 2 and apply no dropout on it.

We build our model in TensorFlow. The Hybrid-CD model is trained via the mini-batch Adam algorithm [23]. We set the batch size as 64, the max length of token sequence as 80. The learning rate was set as 0.01.

IV. Evaluation

In this section, we describe the design of different assessment scenarios for SLAMPA and report on the evaluation results. Specifically, our experiments aim to address the following research questions:

- RQ1: Is Hybrid-CD able to identify similar code snippets?
- RQ2: Given partially written codes, is SLAMPA able to accurately recommend code snippets?

A. Data Set

In this subsection, we will describe how to construct experimental data sets for SLAMPA. We download the BigCloneBench from its website [5] to train and evaluate our models. BigCloneBench is one of the biggest code clone benchmarks publicly available. This benchmark is built by manually labeling clone or non-clone pairs of code snippets on the large scale inter-project repository IJaDataset 2.0 (approximately 25,000 open-source Java projects with 365 millions of lines of code). The true clones in BigCloneBench are tagged as T1, T2 , ST3, MT3, WT3/T4.

[3]https://fasttext.cc/docs/en/english-vectors.html
[4]https://github.com/facebookresearch/fastText
[5]https://github.com/clonebench/BigCloneBench

TABLE I
OVERVIEW OF THE DATA SETS.

Data set	# of Snippets	# of True Clone Pairs					# of False Clone Pairs
		T1	T2	ST3	MT3	WT3/T4	
Raw Data	545,665	16,185	3,787	12,114	55,106	6,158,975	262,465
Filtered	36,736	16,090	3,757	11,313	48,403	0	79,563
Training	9,685	3,218	751	2,262	9,680	0	15,912

TABLE II
RECALL SCORES COMPARISON AMONG MODELS

Model	Clone Types				Total(%)
	T1(%)	T2(%)	ST3(%)	MT3(%)	
Deckard	95	85	80	22	48
NiCad	100	99	81	1	37
SourcererCC	100	97	77	6	39
CCLearner	100	98	92	30	56
Hybrid-CD	100	99	94	51	69

TABLE III
PRECISION SCORES OF HYBRID-CD WITH DIFFERENT THRESHOLDS

Threshold	# of reported clones	# of true T1-MT3 clones	# of true WT3/T4 clones	Total Precision (%)
0.5	710,517	53,443	615,833	94
0.6	450,637	49,737	392,465	98
0.7	261,525	44,103	213,260	98
0.8	137,908	36,037	100,815	99

1) Data Set for Clone Detection: We leverage the tagged clone pairs in BigCloneBench to train Hybrid-CD model. We observed that the semantically-similar snippets (WT3 or T4 clones) could have totally different implementations even of the same functionality [7]. Since Hybrid-CD model is built to retrieve code snippets for recommendation and code reuse, the semantically-similar snippets which have few tokens same with the partially written code are useless. To build a appropriate data set, we only consider syntactically-similar snippets including T1, T2, ST3 and MT3 clone pairs in the BigCloneBench for training. We also took the method-filtering process proposed in [22], which filters out methods containing less than six lines of code to avoid noises caused by small clone methods.

Finally, We get 79,563 pairs of true clone after these processes. In addition, a good clone detection model should be able to accurately distinguish whether two snippets are clones or not, so we randomly choose false clone pairs from the BigCloneBench to construct a equal-sized negative data set. This negative data set is used to control the false positive ratio of our model.

After that, we randomly extract 20% of these clone pairs as a training data set. It should be noted that the entire 79,563 clone pairs set involves 36,736 unique methods, while the training set only contains 9,685 unique methods, which means that most of the methods in BigCloneBench are unseen for our model. We only use a very small amount of data for training in order to demonstrate the generalization capability of our model. The details of the data sets for clone detection is reported in Table I.

2) Data Set for Statistical Language Model: One of the most important step during the process of SLAMPA recommending is statistical language model based intention analyzing. Since we use a deep neural language model as the intention analyzer in this work, we need a appropriate training corpus to the language model. Unlike clone detection, neural language model can be trained in a un-supervised way, i.e., we can use the input of the next time step as the target of the current time step. So we can leverage all the code snippets in IJaDataset excluding those used in clone detection to build the language model. There are 545,665 complete code snippets in our downloaded data, and we filter out methods containing less than six lines and get 283,867 code snippets. We further remove the snippets used in clone detection to prevent our language model from learning those snippets in advance. Finally we obtain 247,131 code snippets.

After that, we randomly extract 80% of these code snippets as a training set and leave the rest 20% of them as a testing set, which is used for neural language model tuning.

B. RQ1: Finding similar code snippets

1) Experimental Setup: Since the ability of identifying code clones reflects the ability of finding similar code snippets, we focus on assessing whether our Hybrid-CD can effectively identify clones in the BigCloneBench in this subsection.

We feed each code snippet referenced in the BigCloneBench to models in order to detect its clones in the benchmark. To quantify Hybrid-CD's performance, we compared our approach with four popular clone detection tools: Deckard [24], NiCad [25], SourcererCC [26], and CCLearner [22]. It should be noted that neural network-based models (including Hybrid-CD and CCLearner) need a little amount of data for training, which means these models have learned some answers in advance. For a fair comparison, the tagged clone pairs used for training are excluded from the evaluation. All these four tools were executed with the default parameter configuration. We set the threshold of Hybrid-CD as 0.7, i.e., two code snippets are reported as a clone if the similarity between them is greater than 0.7.

2) Evaluation Metrics: We define $Recall$ to measure how many tagged true clone pairs in the benchmark are detected

by a clone detection approach, and define $Precision$ to measure how many of the clone pairs reported by a clone detector are actually true clone pairs. Formally:

$$Recall_{CD} = \frac{|D_{exclusive} \cap GT_{exclusive}|}{|GT_{exclusive}|} \quad (10)$$

$$Precision_{CD} = \frac{|D_{exclusive} \cap GT_{exclusive}|}{|D_{exclusive}|} \quad (11)$$

where $D_{exclusive}$ is the true clone pairs detected by a clone detector, and $GT_{exclusive}$ is the tagged true clone pairs (Ground Truth) in the benchmark excluding the clone pairs used for training.

3) Experimental Result: Table II shows the recall scores among models. For T1 and T2 clones, all the approaches have excellent recall ($\geq$ 95%) except for Deckard, which has about 85% recall on T2 clones. For ST3 clones, the two neural network based models CCLearner and Hybrid-CD easily detect more than 90% of the clones, while the other three tools have only about 80% detection ratio. What makes Hybrid-CD stands out are MT3 clones, the largest part of our data set. Most approaches compared in our experiment fail to detect such clones or have low detection ratio. On the contrary, Hybrid-CD detects more than half of the MT3 clones.

We further explore the precision of Hybrid-CD. We are surprised to find that even through Hybrid-CD is trained via T1, T2, ST3 and MT3 clone pairs, this approach has strong generalization capability and is able to precisely detect WT3/T4 clones. We set different thresholds to Hybrid-CD and measure the precision, Table III shows the experimental result. It can be seen from the table that the most of reported clones are true clones in BigCloneBench. If we set the threshold to a appropriate value, Hybrid-CD is capable of detecting almost all the T1-MT3 clones and detecting a large amount of WT3/T4 clones at the same time.

It should be recalled that Hybrid-CD is built to retrieve code snippets for recommendation and reuse, the main responsibility of Hybrid-CD is detecting T1-MT3 clones. So as the threshold grows up, the number of reported WT3/T4 clones rapidly declines but the number of detected T1-MT3 clones remains large.

We can learn from the recall score and precision score that Hybrid-CD is an effective code detection approach. There are two reasons to explain Hybrid-CD's effectiveness. First, Hybrid-CD leverages the state-of-the-art embedding technique fastText and the state-of-the-art deep neural network BiLSTM to model the source code. It measures similarity between code snippets through the joint embedding and deep learning. Second, Hybrid-CD takes advantage of combining automatically extracted high level features and handcrafted low level features, which help Hybrid-CD entirely understand the code from different perspectives.

C. RQ2: Recommending code snippets during programming

1) Experimental Setup: The high-level goal of SLAMPA is recommending appropriate code snippets to prevent programmers from reinventing wheels without knowing the potential reuse opportunities. SLAMPA combines statistical language model and code clone detection techniques to identify similar snippets to the current partially written code in the process of programming.

TABLE IV
OVERALL RESULTS OF SLAMPAS

Model	HR@1	HR@5	HR@10	MRR
SLAMPA-NoLM	0.263	0.416	0.494	0.343
SLAMPA-Base	0.362	0.547	0.654	0.459
SLAMPA-FL	0.384	0.596	0.682	0.481
Abs. Improv.*	0.121	0.18	0.188	0.116
Rel. Improv.	46.0%	43.2%	38.1%	33.5%

* The value of improvement is calculated according to SLAMPA-NoLM and SLAMPA-FL.

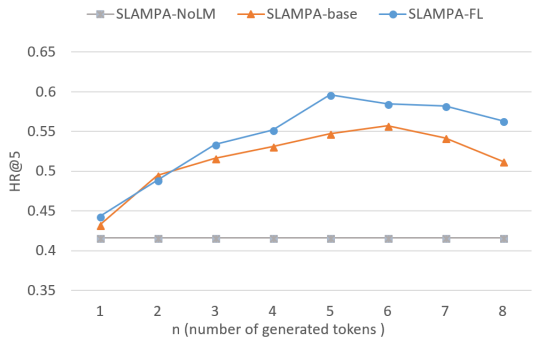

Fig. 7. The influence of the number of generated tokens n on SLAMPA. We adjust n and observe the HR@5. Since the adjustment of n has no effect on SLAMPA-NoLM, so it always has the same value and appears as a straight line.

We use BigCloneBench in another form to evaluate the performance of SLAMPA. We tokenize each method in test set into token sequence by our code analyzer and take the first third of the tokens as partly-written program. If SLAMPA finds its original cloned code snippets leveraging the statistical lanauge model, we can mark it as an effective recommendation. The greatest advantage of using BigCloneBench is that we can automatically verify the quality of recommendation via the manually labeled clone pairs in this benchmark: Given a complete code snippet, its cloned code snippets are literally similar to it (T1, T2, ST3, MT3 clone) or have implemented the same functionality in an alternative way (WT3/T4 clone).

The data used for evaluating SLAMPA are almost the same with the data used for evaluating Hybrid-CD, except that the given snippet is cut to construct partially written code.

2) Evaluation Metrics: In order to automatically verify the performance of SLAMPA and taking the reliability of the results into account at the same time, we use two metrics which are widely used in information retrieval to evaluate the results, namely, $HitRate@k$ $(HR@k)$ and $Mean\ Reciprocal\ Rank$ (MRR).

The $HitRate@k$ measures the percentage of queries for which more than one correct results exist in the top k ranked

results. Formally:

$$HitRate@k = \frac{1}{|Q|}\sum_{q\in Q}\xi(R(q),k) \tag{12}$$

where Q is a set of queries (in our case, partially written codes), $R(q)$ is the set of recommended code snippets of query q, and $\xi(\cdot)$ is a function which returns 1 if the rank of the first hit result is no greater than k and 0 otherwise.

The MRR measures the inverse of the first hit rank. The higher the MRR value is, the better the SLAMPA performs. MRR is calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{q\in Q}\frac{1}{First\ hit\ rank\ of\ R(q)} \tag{13}$$

3) Experimental Result: We implement SLAMPA with a LSTM model and a focal-loss-based LSTM model inside separately, which are denoted as SLAMPA-Base and SLAMPA-FL. SLAMPA targets at predicting and recommending codes according to the current partially written code during the process of programming. To the best of our knowledge, there are no other researches try to cope with this problem (will be further discussed in Related Work). In order to properly evaluate the impact of language model on the recommendation quality, we conduct a controlled experiment by removing the language model from SLAMPA. The prototype without language model is called SLAMPA-NoLM.

When a recommendation is required, SLAMPA firstly tokenizes each given snippet into token sequence by our code analyzer. Then, for SLAMPAs with language model, the first third of tokens are fed into the intention analyzer. Intention analyzer reads the tokens and further generates $n = 5$ tokens as the intention. Finally, code snippet retriever retrieves and recommends code snippets. For fairness, SLAMPA-NoLM recommends k most similar snippets to the input query, while SLAMPAs generate k expanded queries using beam search and then recommend the most similar snippet to each expanded query.

Table IV shows the results of our experiments. The columns HR@1, HR@5 and HR@10 show the results of HitRate@k when k is 1, 5 and 10 respectively. The column MRR shows the MRR values of SLAMPAs.

Language models bring improvements: The results show that SLAMPAs work better than SLAMPA-NoLM. For example, HR@5 value of SLAMPA-FL is 0.596, which means that for 59.6% of the queries, the relevant code snippets can be found within the top 5 recommended snippets. For the HitRate@k, the improvements from SLAMPA-FL to SLAMPA-NoLM are 46.0%, 43.2% and 38.1% respectively. Besides, the MRR value of SLAMPA-FL is 0.481 while the MRR value of SLAMPA-NoLM is only 0.343. The improvement of SLAMPA-FL beyond SLAMPA-NoLM in MRR is 33.5%. The results also show that both SLAMPAs have significantly better performance than SLAMPA-NoLM. We can therefore say that the intention inferred by statistical language model can indeed hit the demands of programmers and the recommended code snippets are effective for code reuse.

```
public void update() {
 String passwordMD5 = new String();
 if (this.password!= null && this.password.length()>0){
     MessageDigest md = MessageDigest.getInstance("md5");
-----------------------cut at here-----------------------
     md.update(this.password.getBytes());
     byte[] digest = md.digest();
     for (int i = 0; i < digest.length; i++) {
         passwordMD5 +=
             Integer.toHexString((digest[i] >> 4) & 0xf);
         passwordMD5 +=
             Integer.toHexString((digest[i] & 0xf));
     }
 }
 ...
}
```

Fig. 8. A snippet (id=15821341 in BigCloneBench, located in project "lmsfm") is cut into two disjoint parts, and the first part is fed into SLAMPA as a query.

```
public static String hash(String toEncripty) {
 MessageDigest md = MessageDigest.getInstance("MD5");
  md.update(toEncripty.getBytes());
  byte[] hash = md.digest();
  StringBuffer hexString = new StringBuffer();
  for (int i = 0; i < hash.length; i++) {
      if ((0xff & hash[i]) < 0x10) hexString.append("0" +
            Integer.toHexString((0xFF & hash[i])));
      else hexString.append(Integer.toHexString(0xFF &
           hash[i]));
  }
 ...
}
```

Fig. 9. A snippet (id=16502662 in BigCloneBench, located in project "logicash") recommended by SLAMPA, which is tagged as a MT3 clone of the snippet shown in Figure 8.

The quality of language models affect the performance: By comparing the results of SLAMPA-Base model and SLAMPA-FL model, we can easily see that although both language models promote the quality of recommendation, the enhanced SLAMPA-FL model works better than SLAMPA-Base. SLAMPA-FL easily outperforms SLAMPA-Base 4% - 9% in HitRate and MRR. This result demonstrates that we can further promote the performance of SLAMPA by upgrading the language model inside.

The number of generated tokens affects the performance: Since SLAMPA relies on its language model, the number of tokens to be generated n has an effect on the performance of SLAMPA. Figure 7 shows that only adding 2-3 referred tokens to the query can significantly improve the recommendation quality. We can further get a better performance by generating a little more tokens (4-6). When n keeps increasing, the recommendation quality goes down slightly. This phenomenon shows that although the intention of programmers can be caught by statistical language model, accomplishing the entire task automatically is impractical in real-world projects (because of cascading error, which can occur in iterative methods [27]). This is the main reason that we only try to develop a programming assistant for snippets recommendation rather than implement an automatic code generator.

D. Example of Code Snippets Recommendation

This subsection illustrates an example of SLAMPA's recommendation. We cut the snippet shown in Figure 8 into two disjoint parts, and feed the first part as the partially written code (i.e., query) into SLAMPA. SLAMPA analyzes the intention of the query and successfully predicts tokens [MessageDigest.update(), MessageDigest.digest(), ...]. The generated tokens help SLAMPA retrieve the snippet shows in Figure 9, which is tagged as a MT3 clone of the original snippet. This result demonstrates that SLAMPA is able to find the potential reuse opportunities and further recommends appropriate code snippets to programmers during programming.

V. Related Work

Currently, there are two strategies for code recommendations: recommendation by generation and recommendation by search. In this section, we will present related works about code generation and code search.

A. Code Generation

In recent years, researchers try to adopt deep learning techniques for code generation. Balog et al. proposed DeepCoder [28] to generate code. DeepCoder leverages Domain Specific Language (DSL) and program synthesis techniques to search and combine DSL into simple programs. Gu et al. [29] apply deep learning techniques for API learning and proposed DeepAPI. DeepAPI semantically generates API usage sequences for a given natural language query using RNNs.

A potential problem of these approaches is that the automatically generated code may not meet the requirements, or even impossible to be compiled.

SLAMPA evades this problem via the combination of intention analyzing and similar snippets retrieving. SLAMPA implicitly generates tokens according to the given partially written code and finally recommends compilable code snippets from high-quality code repository to programmers.

B. Code Search

In code search, there are many approaches proposed. Much of the existing works focus on free-form query searching. CodeHow proposed by Lv et al. [3] and DeepCS proposed by Gu et al. [4] aim to find code snippets relevant to a user query written in natural language. DeepCS [4] incorporates an extended Boolean model and explores API documents to identify relationships between query terms and APIs. DeepCS measures the similarity between code snippets and user queries through joint embedding and deep learning.

Some publicly available code search engines [5][6] identify key words in the given code to find snippets. Unfortunately, these code search engines appear to be able to identify only identical code fragments [7].

There are a few other researches attempting to tackle the code-to-code searching problem. FaCoY proposed by Kim et al. [7] takes a completed code fragment of some functionality as input, and finds semantically similar code fragments to the given fragment. It is used to find alternative implementations of some functionalities.

SLAMPA differs from these existing code search techniques in the following aspects: (1) SLAMPA takes a partly-written code as input, instead of free-form query, keywords or completed code snippet. (2) SLAMPA focuses on mining the intention under the given partly-written code for code recommendation. (3) SLAMPA recommends code snippets during the process of programming, neither before programming (free-form query) nor after programming (completed code).

C. Code Completion

Recently, researchers have investigated possible applications of statistical language model to code completion. Hindle et al. [9] applied a N-gram model to complete code, and Nguyen et al. [30] extended N-gram model by taking semantic information of source code into consideration. Nowadays, neural network techniques such as RNNs have achieved the state-of-the-art results in code completion tasks. Liu et al. [31] proposed a code completion model based on LSTM network and Li et al. [32] extended it with a pointer mixture network.

Most of the existing works focus on next-token suggestion and come in the form of code completion plugins integrated with IDEs. These approaches cannot find completed code snippets.

SLAMPA adopts a simple neural language model and slightly adjusts it to repeatly generate tokens. It leverages the language model and the Hybrid-CD we proposed to retrieve compilable code snippets from high-quality code repository and recommend them to programmers. It does not try to automatically complete the entire program.

It is apparent that we do not try to propose a new approach for building language model or code completion in this paper. Actually, we are looking forward to the development of code completion techniques. As we shown in Section IV that SLAMPA can be further enhanced by introducing more powerful language models.

D. Clone Detection

There are mainly five types of clone detection techniques developed: text-based, token-based, tree-based, graph-based, and metrics-based [33]. Token-based techniques are inherently language-independent and low-cost. They also need less resources, which in turn makes them more scalable [34].

Token-based clone detection approaches first identify tokens and remove white spaces and comments from source code. Then most of them detect clones based on token comparison [25][26]. For example, NiCad [25] replaces identifiers related to types, variables, and constants with special tokens, and then compare the resulting token sequences. SourcererCC [26] indexes code blocks with the least frequent tokens contained in the blocks, and then compares blocks indexed by the same token to find clones.

In this paper, we mainly tokenize source code at API level and leverage deep neural network to measure the similarity. Compared with traditional token-based approaches, our approach is more robust in handling T3 clones.

VI. Conclusion And Future Work

In this paper, we present SLAMPA, a novel snippets code recommending tool with a statistical language model and a hybrid code clone technique named Hybrid-CD. Given a piece of partially written code, SLAMPA can infer its intentions, retrieve relevant code from codebase with the support of Hybrid-CD, and recommend the most similar code snippets to programmers.

Our experiments show: (1) Hybrid-CD outperforms the existing works and is effective enough to retrieve similar code snippets; (2) Leveraging a deep neural language model, SLAMPA is able to find the potential reuse opportunities and recommends appropriate code snippets to programmers during programming; (3) The performance of SLAMPA is related to the capability of language model.

As for our future work, we will further enhance SLAMPA by introducing the latest language models (e.g., Pointer Sentinel-LSTM [35]). Furthermore, we will incorporate some domain specific knowledge (e.g., return type, related document and comments, etc.) into SLAMPA, which are not in the scope of consideration yet in this work.

VII. Acknowledgement

This research is supported by 973 Program in China (Grant No. 2015CB352203) and National Natural Science Foundation of China (Grant No. 61472242).

References

[1] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 664–675.

[2] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, *MAPO: Mining and Recommending API Usage Patterns*. Springer Berlin Heidelberg, 2009.

[3] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 260–270.

[4] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 2018 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[5] "Searchcode," https://searchcode.com/, accessed June 15, 2018.

[6] "Krugle," http://www.krugle.org/, accessed June 15, 2018.

[7] K. Kim, D. Kim, T. F. Bissyande, E. Choi, L. Li, J. Klein, and Y. Le Traon, "Facoy–a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[8] Y. Lin, G. Meng, Y. Xue, Z. Xing, J. Sun, X. Peng, Y. Liu, W. Zhao, and J. Dong, "Mining implicit design templates for actionable code reuse," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 394–404.

[9] 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.

[10] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, 2007.

[12] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 438–449.

[13] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[14] M. Aeschlimann, D. Baumer, and J. Lanneluc, "Java tool smithing extending the eclipse java development tools," *Proc. 2nd EclipseCon*, 2005.

[15] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *Eprint Arxiv*, 2014.

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

[17] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, and K. Macherey, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[18] W. J. Murdoch and A. Szlam, "Automatic rule extraction from long short term memory networks," 2017.

[19] T. Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal loss for dense object detection," pp. 2999–3007, 2017.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[21] 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.

[22] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 249–260.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[25] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[26] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[27] J. Neville and D. Jensen, "A bias/variance decomposition for models using collective inference," *Machine Learning*, vol. 73, no. 1, pp. 87–106, 2008.

[28] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[29] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

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

[31] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016.

[32] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," 2017.

[33] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[34] Y. Yuan and Y. Guo, "Boreas: an accurate and scalable token-based approach to code clone detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 286–289.

[35] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.