Lancer: Your Code Tell Me What You Need

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

***Abstract*—Programming is typically a difficult and repetitive task. Programmers encounter endless problems during programming, and they often need to write similar code over and over again. To prevent programmers from reinventing wheels thus increase their productivity, we propose a context-aware code-to-code recommendation tool named Lancer. With the support of a Library-Sensitive Language Model (LSLM) and the BERT model, Lancer is able to automatically analyze the intention of the incomplete code and recommend relevant and reusable code samples in real-time. A video demonstration of Lancer can be found at https://youtu.be/tO9nhqZY35g. Lancer is open source and the code is available at https://github.com/sfzhou5678/Lancer.**

***Index Terms*—Code recommendation, Code reuse, Language model**

I. Introduction

In software development activities, when programmers encounter problems, they often search code examples to learn how to handle such problems [1], [2]. Researchers [3], [4] have proposed some code search engines to retrieve code samples, and some recent approaches [5], [6] even allow free-form queries (*i.e.*, how to generate an md5 hash) to search relevant code snippets. Alternatively, some clone detection techniques [7], [8] can locate similar code snippets of a given code sample. They allow programmers to refine their code by learning its clones.

Although the above tools are useful, existing tools are still not smart enough to recommend relevant code samples during programming in real-time. For example, when a novice write the incomplete code shown in Figure 1, she may not know how to continue the programming task, and needs code samples to learn how to complete it. Since she is a novice and do not know the details, it is hard for her to use only natural language to accurately depict her intention. On the other hand, as the code is incomplete, clone detection techniques often cannot locate useful code samples. As a result, she has to find the solution by herself, which is time-consuming and error-prone. Here, even if an experienced programmer knows that a relevant code sample is useful, she can fail to remember its location, and has to reinvent the wheel. Therefore, locating appropriate code examples is still a serious challenge for programmers.

To handle these challenges, we propose a recommendation tool named SLAMPA in our previous work [9], which can recommend code samples based on what is already written under development. However, as SLAMPA uses only deep learning technologies, it has the out-of-vocabulary (OOV) problem [10]. In addition, its accuracies are relatively low, and it is not fast enough to support real development.

```
public ImageData getJPEGDiagram() {
  Shell shell = new Shell();
  GraphicalViewer viewer=new ScrollingGraphicalViewer();
  LayerManager lm = (LayerManager) viewer.
        getEditPartRegistry().get(LayerManager.ID);
  ...
}
```

Fig. 1. A piece of incomplete code.

In this paper, we proposed Lancer to further improve the state of the art. Compared with SLAMPA, Lancer presents a new language model called Library-Sensitive Language Model (LSLM), and new algorithms to rank samples. We constructed 2892 programming tasks, in which tools shall recommend related code for a piece of incomplete code. We compare Lancer with SLAMPA and two other clone detection tools such as SourcererCC and CCLearner, for their effectiveness of recommending related code samples. For 45.4% of our programming tasks, the first recommended code sample of Lancer is related to the incomplete code, while the compared approaches can only handle $8.7 \sim 25.3\%$ of them. In addition, on average, our result shows that Lancer recommends a code sample in 0.67 seconds, which is fast enough for real development.

II. Lacner

Figure 2 shows the overview of Lancer. We implement its UI as an IntelliJ IDEA [11] plugin. A complete method in our code repository is called as a code sample. The plugin will recommend related code samples to programmers, based on their incomplete code, when they are coding. Lancer supports Java for now. To prepare the code repository for recommendation, we implement a crawler to download source files from open source communities (*e.g.*, Github). In total, we downloaded 12,674 Java projects which contains 16 million code samples[1]. To support recommending code samples, Lancer implement a back-end server. We next introduce its components.

A. Code Parser

We implement a method-level code parser atop of SLP-Core [10], and it parses incomplete code under development and source files in our code repository. From source code, it extracts key information such as libraries, class

*Corresponding author.

[1]To make a fair comparison with other tools, in Section III, we temporally replace the repository with a much smaller benchmark.

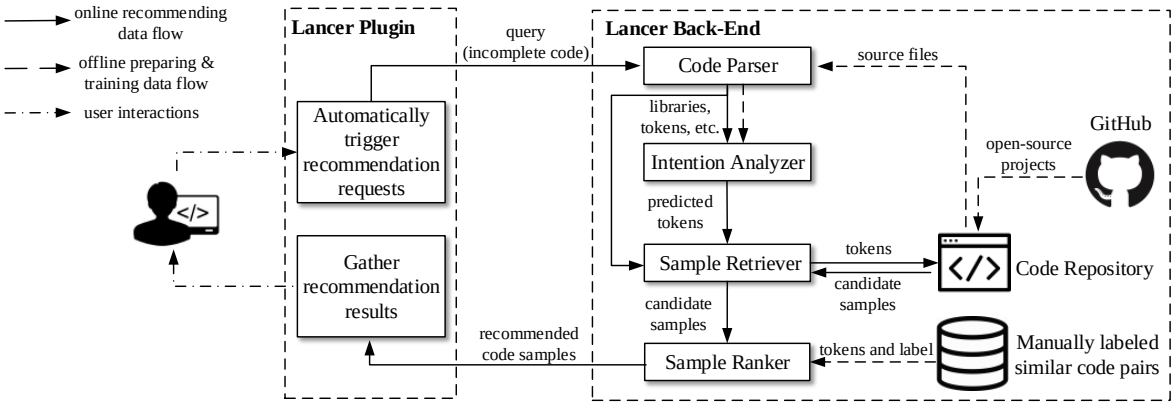

Fig. 2. The overview of Lancer.

names, method names, token sequences, return types, and parameter types for each method. Compared with SLAMPA, Lancer models methods in token-granularity rather than API-granularity to alleviate the OOV problem.

B. Intention Analyzer

Hindle et al. [12] reported that language models can capture the the repetition of programming languages (*i.e.* code pattern). As shown in Equation 1, a language model estimates the probability of a sequence $S = t_1, t_2, ..., t_m$.

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

where $Pr(t_i|t_1, ..., t_{i-1})$ is the probability of occurrence of each token when given its preceding tokens.

Given the token sequence of a parsed incomplete method, Lancer captures its intention with our Library-Sensitive Language Model (LSLM). With its support, Lancer is able to predict follow-up tokens for an incomplete method. The predicted tokens are implicitly appended to the current tokens thus produce a semantically consistent and more complete token sequence. Lancer then uses these tokens to retrieve code samples (Section II-C) which are highly related to the given incomplete method, and contain the code that the programmers need to write as well.

Library-Sensitive Language Model. It is insufficient to directly learn a language model from the parsed source code as SLAMPA and other previous works did [10], [12], since it can lose the details of programming scenarios. For example, the token $SWT.Text$ is much more likely to be invoked in Java Graphical User Interface (GUI) related projects than its overall probability, which is worth of consideration.

For a given code sample, we propose Library-Sensitive Language Model (LSLM) to identify its application scenario. LSLM is built on the open-vocabulary cache N-gram model that is proposed by Hellendoorn and Devanbu [10], and we extends this model to capture the code patterns for each library (*e.g.* org.eclipse.jdt) separately. We select this model, since Hellendoorn and Devanbu [10] report that it less suffers from the OOV problem. In particular, it counts tokens within a local scope and combines it with the general counter to achieve the final prediction.

Our N-gram model for each library is initially learnt from the tokens in source files which contain the specific library. These models can also dynamically learn code pattern in practice to alleviate the OOV problem. Compared with the deep learning based language model used in SLAMPA, the open-vocabulary cache N-gram model has a faster speed and it is more robust when encountering OOV words [10] .

As multiple libraries are usually used together in a certain scenario (*e.g.* SWT and $Swing$ are usually used together in Java GUI projects), we further leverage Topic-Sensitive PageRank [13], a graph-heuristic association mining algorithm, to assess the relevance scores between libraries. When an incomplete code and the corresponding parsed tokens come, LSLM finds out the most relevant libraries to the code and predicts the several subsequent tokens. For each token prediction, LSLM combines the predicted probabilities of all the language models (one LM per relevant library) according to their relevance scores to get the final probabilities, as shown in Equation 2. The token with highest probability is selected as the current predicted token.

$$Final\ Probability = \sum_{i=1}^{L} w_i * prob_i \quad (2)$$

where L is the number of relevant libraries, w is the relevance score and $prob$ is the predicted token probabilities of each language model.

C. Sample Retriever

This component builds inverted indices for parsed code samples. When given a token sequence as the query, Lancer has a coarse-grained matching phase to reduced the entire code repository into hundreds of candidate samples to greatly improve the retrieval efficiency, which is different with SLAMPA's pairwise comparison.

1) Indexing: Lancer uses Elasticsearch [14] to index parsed code samples. Elasticsearch is a distributed search engine built for the cloud. Lancer builds inverted indices for tokens of all the fields (*e.g.* class/method names, token sequences, etc.) of each parsed code sample. The inverted

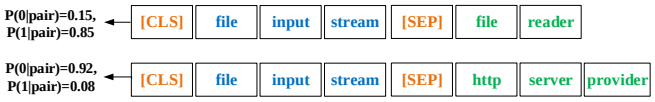

Fig. 3. An illustration of the BERT-based Deep Semantic Ranking Scheme. We pack the keyword sequences of the given method (in blue) and each candidate code sample (in green) into a single sequence seperatly.

index of a particular token records which code samples contain the token.

2) Matching: Given the tokens of an incomplete method and the tokens predicted by our intention analyzer, Lancer utilizes the built inverted indices to get code samples which are related to these tokens from our enormous code repository. Then Lancer uses the BM25 algorithm to estimate the relevance of these code samples according to tokens. The top one hundred samples are considered as candidate samples to be recommended.

3) Filtering: Lancer takes an extra step to remove the duplicate samples. We convert the token sequences of candidate samples into token-frequency dictionaries D. We then calculate similarity scores between two code samples (s_A and s_B) as follows:

$$Sim(s_A, s_B) = 1 - \frac{\sum_x |freq(D_A, x) - freq(D_B, x)|}{\sum_x |freq(D_A, x) + freq(D_B, x)|} \quad (3)$$
$$x \in tokens(D_A) \cup tokens(D_B)$$

where $freq(D_A, x)$ is a function which returns the frequency of token x in the token-frequency dictionary D_A. When two sequences have no overlapped tokens, the similarity score will be set as 0.5 by default. If the similarity between one sample and any other sample exceeds threshold $\sigma = 0.9$, the sample will be removed.

D. Sample Ranker

If Lancer finds more than one candidate sample for a given tokens, it uses a ranker to sort the samples. It is worth noting that simply assuming the incomplete code given by programmers and all the predictions made by our intention analyzer are extractly correct may damage the robustness of our code recommendation tool. Different with SLAMPA, Lancer's deep semantic ranking scheme, which leverages the state of the art technique BERT [15] for text representation, is able to filter out the noise candidate samples.

Deep Semantic Ranking Scheme. Typically, code names reflect their functionalities [16]. BERT is a bidirectional pre-training language model for text representation which has achieved dazzling success in a large number of scenarios. We incorporate BERT to leverage the code name information. We finetune a pre-trained BERT model[2] on manually labeled similar code pairs, and the trained model can rank code samples from a semantic perspective. Given a piece of incomplete method and a set of code samples, we first split their code names (*i.e.*, class names, method names, and variable names) into word sequences by their camel names. For example, we split "$InputStream$" into "$input$", and "$stream$". As shown in Figure 3 that we then pack the words sequences of the given method and each sample with special tokens $[SEP]$ and $[CLS]$ separately. The token $[SEP]$ is used to separate the two input sequences. The final hidden state of the token $[CLS]$ is used as the aggregate sequence representation for classification tasks. The integrated sequence is finally fed into BERT for binary classification, where label 1 stands for relevant while label 0 is on the contrary. The probability of being classified as label 1 is used as the final semantic correlation score for each pair.

[2]https://github.com/huggingface/pytorch-pretrained-BERT

III. Evaluation

In our evaluation, we compared Lancer with SourcererCC [8], CCLearner [17], and SLAPMA [9], for their effectiveness of recommending code samples.

A. Training Models

1. Dataset. We select the BigCloneBench benchmark [18] as the dataset of our evaluation. The benchmark has 42,120 source files, which includes 301,537 methods, and 8 million LOCs in total. According to their functionalities, the source code files in this benchmark are divided into ten categories (*e.g.* decompressing zip archives). Their clones are manually marked as T1, T2, ST3 (Strong Type 3), MT3 (Moderately Type 3), and WT3/4 (Weak Type 3 or Type 4). Here, T1 denotes identical clones, and T4 denotes semantic clones that are syntactically different. From BigCloneBench, we select all the T1, T2, ST3, and MT3 clones, which include 42,120 files with 79,563 clones.

2. Our repository and our models. To align our inputs with the compared approaches, we replace our GitHub-based code repository with the source files of our dataset. When training our models, as Li et al. did [17], we select the 23,193 source files under the "copy file" category as our training set, and use the remaining 18,927 source files as a test set to construct our tasks.

As our Language model does not need labels, we use all the source files of the training set, when we train the model. Our semantic ranking model needs labels, and we use the clone relations in the training set as the labels.

B. Evaluation Setup

1. Our tasks. We construct programming tasks with clones in our test set. If a method has a clone, we take the top **1/5** code snippet as the input to retrieve its clones. Our tasks illustrate the situations when a developer write some code lines, but fail to complete the remaining ones. We find that 2892 methods of the 18,927 source files have their clones. As a result, in total, we construct 2892 tasks.

2. Evaluation metrics. For each task, we consider its incomplete code as the code under development, and the clones of the incomplete code as the golden standard. If a recommended code sample is the clone of the code under development, we mark it as a success.

We use HitRate@k (HR@k) and Mean Reciprocal Rank (MRR), two metrics widely used in information retrieval,

TABLE I
OVERALL PERFORMANCE OF LANCER AND THE RELATED APPROACHES

Model	HR@1	HR@5	HR@10	MRR	Avg Time (s)
SourcererCC	0.087	0.191	0.240	0.130	-*
CCLearner	0.198	0.289	0.325	0.214	8.23
SLAMPA	0.253	0.368	0.460	0.281	12.73
Lancer	0.454	0.595	0.629	0.515	0.67

* The consumed time of SourcererCC is incalculable, because we execute it several times with different thresholds.

to evaluate the effectiveness. The HitRate@k measures the percentage of queries for which more than one correct results exist in the top k ranked results. Formally:

$$HitRate@k = \frac{1}{|Q|} \sum_{q \in Q} \xi(R(q), k) \tag{4}$$

where Q is a set of queries, $R(q)$ is the set of recommended code samples 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. HitRate@k measures the ability of code recommenders to identify relevant code samples. A better code recommender should allow programmers to get the relevant sample by inspecting fewer returned results. The higher the HitRate value, the better the recommendation performance. The MRR measures the inverse of the first hit rank. MRR is calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{First\ hit\ rank\ of\ R(q)} \tag{5}$$

The higher the MRR value, the better the performance.

SourcererCC does not provide the similarity scores of the reported similar code samples so we cannot directly rank its recommended samples. To address this problem, we merge the results of SourcererCC with different similarity thresholds to approximately get the ranked samples list. All the evaluated models use the same dataset and take the first fifth of the tokens as a query to search relevant code samples.

Besides the above two measures, we record the average execution time to compare the performance of these tools.

C. Evaluation Results

As shown in Table I that Lancer finds more relevant code samples than other approaches. The HR@1 value shows that for 45.4% queries, the first code sample recommended by Lancer is highly relevant to the query. The HR@10 value shows that for 62.9% of the queries, the relevant samples can be found within the top 10 results recommended by Lancer. The results also demonstrate that the recommendations made by Lancer are extremely fast. Benefited from the coarse-grained matching phase described in Section II-C, it only takes 0.67s in average for Lancer to accomplish a recommendation. While traditional clone detectors including SourcererCC and CCLearner have a poor effectiveness in HR and MRR metrics, and it is also time-consuming for these techniques to perform code recommendation. These results demonstrate that traditional clone detection techniques are insufficient to recommend related code samples, when the code under development is incomplete. Although SLAMPA is designed to support this purpose, due to the limited pairwise comparison matching mechanism and the OOV problem encountered in practice, SLAMPA is much slower than Lancer, and its accuracies are also poorer than Lancer.

IV. CONCLUSION AND FUTURE WORK

In this paper, we present Lancer, a context-aware code-to-code recommending tool leveraging a Library-Sensitive Language Model and a BERT model to recommend relevant code samples in real-time. Our evaluation results show that Lancer is more effective than the compared approaches. In future work, we plan to further explore the following issues: (1) supporting more programming languages than Java; (2) collecting and analyzing feedbacks from developers; (3) training a unified LM model and customizing it for specific libraries; and (4) evaluating more technical details (*e.g.*, the impact of BERT).

V. ACKNOWLEDGEMENT

This research was sponsored by National Key R&D Program of China (Project No. 2018YFB1003903 and 2018YFC0830500), and National Nature Science Foundation of China (Grant No. 61472242 and 61572313).

REFERENCES

[1] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proc. ICSE*. ACM, 2014, pp. 664–675.
[2] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proc. ECOOP*, 2009, pp. 318–343.
[3] "searchcode," https://searchcode.com/.
[4] "Krugle," http://opensearch.krugle.org/.
[5] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on API understanding and extended boolean model," in *Proc. ASE*, 2015, pp. 260–270.
[6] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. ICSE*, 2018, pp. 933–944.
[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 *Proc. ICSE*, 2018, pp. 946–957.
[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proc. ICSE*, 2016, pp. 1157–1168.
[9] S. Zhou, H. Zhong, and B. Shen, "SLAMPA: Recommending code snippets with statistical language model," in *Proc. APSEC*, 2018, pp. 79–88.
[10] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. ESEC/FSE*, 2017, pp. 763–773.
[11] "Intellij idea," https://www.jetbrains.com/idea/.
[12] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. ICSE*, 2012, pp. 837–847.
[13] T. H. Haveliwala, "Topic-sensitive pagerank," in *Proc. WWW*, 2002, pp. 517–526.
[14] "Elasticsearch," https://www.elastic.co/products/elasticsearch.
[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[16] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proc. ECOOP*, 2009, pp. 294–317.
[17] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proc. ICSME*, 2017, pp. 249–260.
[18] 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 *Proc. ICSME*, 2014, pp. 476–480.