PCC:Practical Code Completion Tool with Fuzzy Matching and Precise Synthesis on Statistical Language Modelling

Yixiao Yang
Tsinghua University, TNLIST, KLISS, Beijing, China
yangyixiaofirst@163.com

ABSTRACT

As open source has become a trend, more and more good projects can be found on github. In countless projects, much of the code can be very similar. But another fact is that in most cases, the already existed code can not be directly copied and pasted. Besides, searching for similar code is difficult, especially for structurally similar one.

We present PCC, a plugin of eclipse IDE that analyzes the already written code, uses the sequence fuzzy matching algorithm to find similar contexts, predicts the most likely sequence based on the pre-trained statistical language model and automatically synthesizes the code from the sequence.

PCC synthesises a whole statement which may include all syntax elements of Java. PCC takes the context of currently being edited java file into consideration and ensures that there is no compilation errors except the unresolved method invocations in the generated code.

The correctness is 95% among 100 test cases. The time cost of 80% of the completion is less than 1 second, which is efficiently and user-friendliness. In the worst case, the time cost is still less than 3 seconds. To best of our knowledge, it is the first time to apply fuzzing matching algorithm to match and predict the code patterns.

CCS Concepts

•Software and its engineering → Search-based software engineering;

Keywords

Code Completion, Code Prediction, Synthesis, Automatically Code Writing, Statistical Language Modelling

1. INTRODUCTION

Since 2012, researchers have tried to apply different machine learning algorithms to software engineering. When applying algorithms of natural languages to source codes, things become more complicated. Unlike natural languages, source codes are closely related. Very minor modifications may cause compilation errors. This brings challenges to code automatic generation.

PCC synthesizes codes based on patterns which is different from Test Driven Synthesis technique which is based on genetic algorithms. There have been multiple code synthesis and fix tools[12][6][8] driven by genetic algorithms. These tools can generate very accurate code based on oracle test cases. However, there is still a long way to achieve the goal of automatic code writing. Although the integer expression synthesis and the boolean expression synthesis have been developed for a long time, the problem synthesizing with external method invocations is still very difficult. There would be too many possible situations. What is more, the genetic algorithm is slow. Scale is a concern.

To narrow the whole possible search space, applying machine learning algorithms to big-code to learn how to write codes has become an effective method.

1.1 Basic Ngram Model

Because the improvements brought by different n-gram algorithms are very small and the fuzzing matching with precise synthesis is the vital concern, PCC uses the basic n-gram model.

Training Process computes all the conditional probability:
$P(s_m|s_{m-1}...s_{m-(n-1)}) = \frac{c(s_m...s_{m-(n-1)})}{c(s_{m-1}...s_{m-(n-1)})}$
where c(·) is the count of the given n-gram.

After training, Given a sequence of tokens $s_1...s_{n-1}$ and an available n-gram model, the probability of next token is:
$NP(s_{next}) = P(s_{next}|s_1...s_{n-1}) * F(s_1...s_{n-1})$
$F(s_1...s_{n-1}) = \prod_{m=1}^{n-1} P(s_m|s_{m-1}...s_{m-(n-1)})$
To infer the next token with the n-gram model, the longest context is the last n-1 tokens.

2. RELATED WORK

The first one to apply natural language processing algorithms to source codes was Abram Hindle[4], he used the n-gram[2] model to train the source codes of Java. Next work was done by Miltiadis Allamanis[1], he trained 100 times larger model than Hindle and made a topic visualization tools to show the frequently used codes of the software. Tung Thanh Nguyen[7] refined the tokens, used the n-gram topic[11] model, did the experiments similarly.

The predecessors have done great works, but their model can hardly be used to predict and synthesize the codes. The reason is that their models are too scattered. For example, String str=a.toString() will be split into 8 tokens:String#str #=#a#.#toString#(#). Every word split by # is a token. If a 5-gram model is available, when inferring the next token from the above example, what will be inferred? Now the longest context is the last 4 tokens:.#toString#(#). Even with the longest context, any token behind statements such as String str=a.toString(), String str=b.toString(), String any=anything.toString() and so on will show up. Information before .toString() is lost. According to our experiments, if the model is large, about 100 different tokens with similar probabilities will be got. Choosing the next token has become a problem. What is more, different people have their

own name conventions. Some people like to declare variable like this:String ssssss1 = a.toString().

The model can not be compact either. Due to database being searched by the exact key, if taking the statement String str=a.toString() as one token, nothing will be inferred from the statement String str=b.toString() although there is only one character different between two string tokens.

Therefore, to make a practical code completion tool, an intermediate representation of java codes is introduced and the fuzzing matching algorithm is used.

Recently, Anh Tuan Nguyen[5], Veselin Raychev and Martin Vechev[9] used statistical language model to do the theoretical research on API usages.

3. ARCHITECTURE

Figure 1 shows the architecture and the execution flow of the tool.

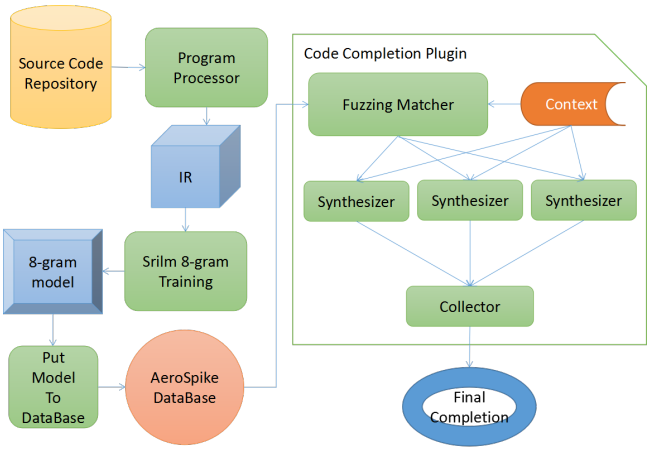

Figure 1: PCC architecture

Source Code Repository contains java source files crawled from github and other websites.

Program Processor translates java source files to IR.

IR is a special form which meet the condition: one operation one token. The basic idea is that if a statement is too long, we split it into several tokens. IR is carefully designed to ensure these separated tokens can be rebuilt to the original statement. Variable names are eliminated in IR.

Due to one token contains only one operation, it is very easy for IR to do sequence fuzzing matching.

As discussed in Related Work, IR can not be too scattered nor too compact. After trying for 2 months, a suitable kind of IR was finally found.

```
List<String> list = new LinkedList<String>();
Iterator<String> itr = list.iterator();
        ⇩
VD@List VH@=@HO MI@LinkedList(new);
VD@Iterator VH@=@HO MI@iterator(@C0?0);
```

Figure 2: translate to IR

Firgure 2 gives a concrete idea about the translation and the IR. VD@ means a type declared. VH@ means a variable declared. @HO means some operations are after this token. MI@ meas method invocation.

Notice that, In figure 2, the strange symbol C0?0 refer to variable list which is of type List<String>.

In IR, every word split by white spaces or line breaks is a token. So there are three tokens in the following line:
VD@List VH@=@HO MI@LinkedList(new);

Figure 3 shows the translation rule of the variable name.

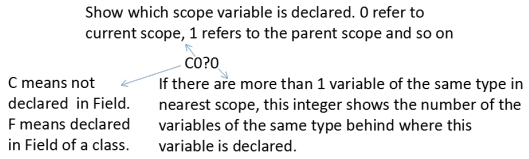

Figure 3: variables translate to IR

Figure 4 shows more examples about the IR translation.

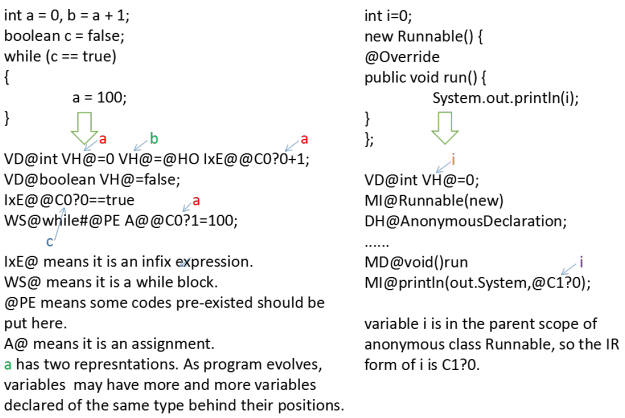

Figure 4: more examples to IR

Then the IR is trained by Srilm[10] tool which is a famous machine learning library in the field of natural language processing. The maximum capacity of Srilm[10] tool is to train 9-gram models.

In order to preserve the information as much as possible, the 8-gram model is trained.

After the 8-gram model been trained, it is put into Aero Spike[3] which is a mature distributed key-value database.

When users are editing java files in eclipse and press alt+/ or other shortcut keys to invoke content assistance, the PCC code completion plugin will extract the context of the currently edited java file, use the program processor to parse the context to IR, put the IR to the Fuzzing Matcher. The possible sequences returned by Fuzzing Matcher will be put into multiple Synthesizers to generate the final completions.

Every synthesizer runs in a separated thread. The Collector will collect all the results of all threads and append the results to the tail of the results returned by the internal content assistance plugin of the eclipse.

4. USE OF PCC

5. FUZZING MATCHER

6. PRECISE SYNTHESIZER

7. IMPLEMENTATION

8. EMPIRICAL RESULTS

9. LIMITATION AND FUTURE WORK

10. CONCLUSIONS

We present PCC, a plugin of eclipse IDE, to automatically complete the code when users are writing java codes in eclipse. PCC does not influence the existing functionality of eclipse.

We make a careful analysis of the advantages and disadvantages of the existing work, put forward an IR in which one token represents one operation. Different name conventions of variables are translated into an unified IR form.

The fuzzing matching algorithm is applied to cope with possible changes in codes. The synthesizer involves runtime type-checking to ensure there is no compilation errors except some unresolved method invocations.

PCC is a real-time tool with quick responses. It is very suitable to train some targeted sets according to different users and different projects. Users may find that many codes do not need to be written by themselves.

11. ACKNOWLEDGMENTS

Thanks Professor Song for reading this paper, and offering valuable advice. Thanks Han Liu for his objective points of view on the topic of this paper. Thank Shuhao Zhang for encouraging me to complete this long cycle of work.

12. REFERENCES

[1] M. Allamanis and C. A. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216, 2013.

[2] W. B. Cavnar, J. M. Trenkle, et al. N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

[3] J. Dillon. Aerospike documentation. http://www.aerospike.com/docs/, 2016.

[4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847, 2012.

[5] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 858–868, 2015.

[6] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[7] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 532–542, 2013.

[8] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *ACM SIGPLAN Notices*, volume 49, pages 408–418. ACM, 2014.

[9] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 44, 2014.

[10] A. Stolcke, J. Zheng, W. Wang, and V. Abrash. Srilm at sixteen: Update and outlook. In *Proceedings of IEEE Automatic Speech Recognition and Understanding Workshop*, page 5, 2011.

[11] X. Wang, A. McCallum, and X. Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 697–702. IEEE, 2007.

[12] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009.