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 fuzzy 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[18][10][13] 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 fuzzy 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 sequence of tokens.

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[7], 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 tool to show the frequently used codes of the software. Tung Thanh Nguyen[11] refined the tokens, used the n-gram topic[17] 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 fuzzy matching algorithm is used.

Recently, Anh Tuan Nguyen[9], Veselin Raychev and Martin Vechev[14] 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.

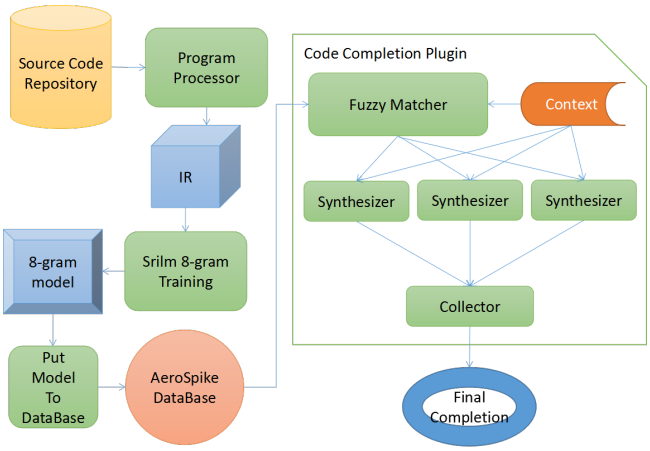

Figure 1: PCC architecture

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 fuzzy 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.

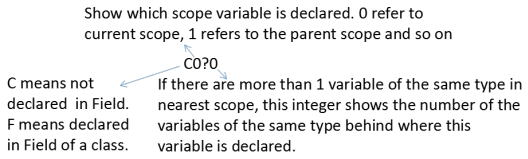

Figure 3: variables translate to IR

Figure 4 shows more examples about the IR translation.

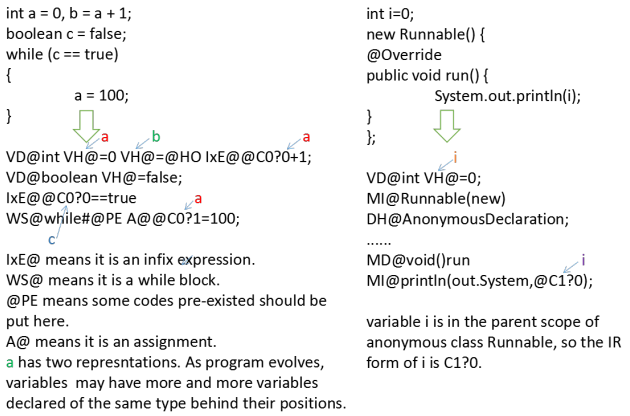

Figure 4: more examples to IR

The grammar of the IR is available at [20]. The program which can parse the IR is available at [21].

Then the IR is trained by Srilm[15] tool. If the Srilm[10] tool is told to train 3-gram models, the 2-gram model and 1-gram model will also be trained. So, if the 8-gram model is trained, the 1-gram to 7-gram models are also be trained.

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[5] 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 Fuzzy Matcher. The possible sequences returned by Fuzzy 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 internal content assistance plugin of the eclipse.

4. PREPARATION OF PCC

Before using PCC, an AeroSpike database which is properly configured must be available.

The program of translating Java to the IR is available at [24]. The tutorial about how to use the translating program is at [25].

The bash script of training the IR to the 8-gram model is available at [22]. The tutorial about how to use the training script is at [23].

The program of putting the 8-gram model into the Aero Spike database is available at [26]. The corresponding tutorial is available at [27].

A crawler which can crawl projects from github is also available at [19]. The format of the files to be downloaded is zip. A script to unzip these zipped files is available at [28].

5. USE OF PCC

The installation way[16] of the PCC tool is the same as the common plug-in[6] of eclipse[4]. When the AeroSpike databases filled with the 8-gram model is available, users need to set the ip of one machine of the AeroSpike clusters on the preference page of their own eclipse. Figure 5 shows the GUI of the preference page.

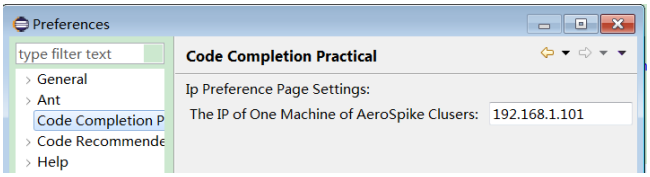

Figure 5: GUI of ip preference setting

When everything is ready, the only thing to do is to press the hot key of content assistance of eclipse which is usually Alt+/ to invoke the internal content assistance plugin and the PCC.

Figure 6 shows the screenshots of the running eclipse with the PCC installed.

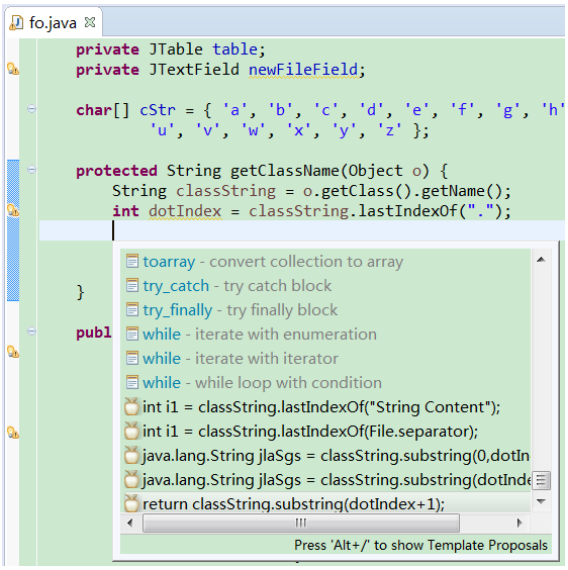

Figure 6: Running Screen Shot

The proposals prefixed by the apple icon are generated by the PCC tool. The lower the position, the higher the priority.

6. FUZZY MATCHER

The fuzzy matching algorithm is combined with the genetic[8] algorithm and the longest common subsequence LCS[3] algorithm.

Figure 7 gives a concrete idea about how this algorithm is running. For the sake of simplicity, the complex IR tokens are replaced by the English alphabets.

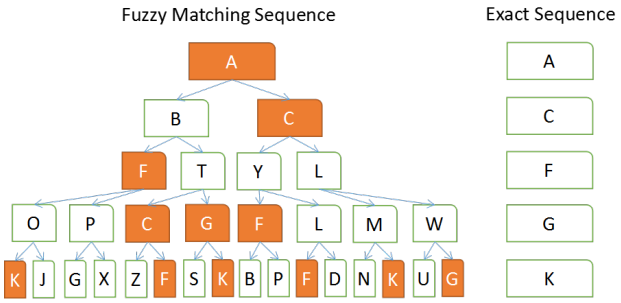

Figure 7: Fuzzy Matching Algorithm

Assume the already written codes are A C F G K. The fuzzy algorithm starts at A, infers the next generation of A based on 2-gram model and computes the longest common subsequence between the two newly generated sequences:AB, AC and the already written codes:ACFGK. Then, the fuzzy algorithm does the similar things to each token of the newly inferred:B, C, but this time, 3-gram model is used because the length of the context is 2. Keep doing the algorithm until the length of the newly generated sequence is more than the length of the already written codes. Note that, 1-gram to 7-gram models are trained internally when the Srilm tool is training the 8-gram model.

The most similar sequences will be retained which are the sequences with orange nodes on its path in figure 7.

7. PRECISE SYNTHESIZER

PCC puts every sequence returned by the Fuzzy Matcher to an unique Precise Synthesizer. Based on the given sequence, the Precise Synthesizer keeps inferring and synthesizing the next token until the stop condition is meet. Each Precise Synthesizer runs in a separated thread. Algorithm 1 shows the framework of the synthesizer.

The function SynthesizeCode is a recursive function which is based on depth-first search. SynthesizeCode has a parameter which represents the current sequence of tokens.

The function InferTokens is used to infer the next tokens from the given sequence.

For each inferred token, the bound needs to be found. For example, given the IR tokens: $mi()$ $@PE\&\&p()$, the bound of token: $@PE\&\&p()$ is $mi()$ because $mi()\&\&p()$ is a whole.

The function Synthesize involves the type checking to ensure that there is no compilation errors.

The internal code completion in eclipse could infer the specifications based on literals. This functionality is integrated into PCC. For example, the internal code completion could translate String to java.lang.String or translate

a.subStr to a.substring(int beginIndex) and a.substring(int beginIndex, int endIndex).

Due to the symbol of the variable name such as C0?0, F0?1 does not contain the information about which type this symbol represents. Therefore, all the types must be tried to synthesize the code. For example, assume two variables declared:int a = 0;boolean b = false, when synthesizing the code:C0?0<1, a<1 and b<1 are tried and b<1 is eliminated due to the type conflict.

Algorithm 1: *SynthesizeCode*(*contextsequence*)

```
Require: contextsequence! = null
  nextgenerations ← InferTokens(contextsequence)
  for token : nextgenerations do
    boundtoken ← FindBound(token)
    success ← Synthesize(boundtoken, token)
    if !success then
      return
    end if
    if CouldStop(contextsequence + token) then
      AppendResult(Synthesize(null, token))
      return
    end if
    SynthesizeCode(contextsequence + token)
  end for
```

8. IMPLEMENTATION

PCC is implemented in Java. The total codes of PCC is 28555. The Antlr4[12] library is used to parse the special IR. The model is stored in memory. The AeroSpike database is configured to use memory as the storage device. The AeroSpike database is installed on a Linux machine. The eclipse with PCC installed is running on a Windows machine. The computers used in experiments all have 16G memory and a 2-core i7 CPU.

9. EMPIRICAL RESULTS

By learning different projects from github, we found that if two projects deal with different areas, only a very small part is similar. Therefore, to simplify our experiments, we only choose projects which deal with similar areas.

Due to the strong type checking of PCC, the project must import the necessary jar packages, otherwise wrong codes may be got. In order to reduce the workload of configuring the jar package in different projects, we choose the projects that do not require or require very little external jar packages.

After screening by the two conditions just mentioned, 2187 java files are selected to do the experiments. Due the small size of the model, we put the model into memory.

If the last five proposals returned by PCC contain the right answer, we think the completion is right. Remember that for proposals, the lower the position, the higher the priority.

We choose 10 different functions including lambda expressions in Java8. For each function, we randomly choose 10 positions. At each selected position, we invoke the content assistance. We compare the proposals returned by PPC with the actual statement following the position where we invoked the content assistance. The 100 test cases consist of these 100 positions.

The experiments show that the accuracy is 98%. Only two test cases are not passed. One failed test case is due to too many possible choices when inferring from 7-gram. Another is due to the failure of choosing the right variable from three variables which all pass the type checks.

In order to evaluate the performance of the fuzzy matching algorithm, we randomly add and delete one or two statements of the context of the 100 test cases, compare the proposals returned by PPC with the actual statement.

The experiments show that the accuracy is greater than or equal to 95%. If we randomly add some statements to the context of the 100 test cases, there is no influence to the accuracy. However, if we randomly delete some statements of the context of the 100 test cases, the accuracy decrease to 95%. The reason for the decline in the correct rate is that the genetic algorithm fails to infer some vital tokens in the process of producing the next generation.

10. LIMITATION AND FUTURE WORK

PCC automatically writes the codes if the completion context is similar to some codes in the database. The computation of the similarity is mainly based on comparing the string contents of each token. If two programs are similar in logic but not similar in structure, PCC will take them as two different programs. What is more, if two programs are structurally similar, but their method names are totally different, PCC will not generate the right code either. When the model becomes more and more large, the genetic algorithm may fail to infer some vital tokens, which leads to the bad results. The future work will mainly focus on generating the logically similar codes.

11. 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 form.

The fuzzy 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.

12. 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.

13. 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] C. E. R. R. L. S. C. Cormen, Thomas H. (EDT)/ Leiserson. *Introduction to Algorithms 3rd Edition*. Mit Press, LCS algorithm, 2009.

[4] I. Corporation. Eclipse help platform. http://help.eclipse.org/mars/index.jsp, 1996.

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

[6] D. R. Eric Clayberg. *Eclipse Plug-ins (4th Edition)*. Addison-Wesley Professional, Introduction, 2012.

[7] 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.

[8] M. Melanie. An introduction to genetic algorithms. www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf, 1996.

[9] 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.

[10] 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.

[11] 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.

[12] T. Parr. Antlr4. http://www.antlr.org/, 2016.

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

[14] 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.

[15] 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.

[16] venukb. Eclipse plugin installation. http://www.venukb.com/2006/08/20/install-eclipse-plugins-the-easy-way/, 2006.

[17] 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.

[18] 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.

[19] Y. Yang. The github crawler. https://github.com/yangyixiaof/gitcrawler/tree/master/gitcrawler, 2016.

[20] Y. Yang. Ir grammar. https://github.com/yangyixiaof/specialparse/blob/master/specialparse/src/main/java/cn/yyx/parse/szparse8java/Java8.g4, 2016.

[21] Y. Yang. Ir parser. https://github.com/yangyixiaof/specialparse/tree/master/specialparse, 2016.

[22] Y. Yang. N-gram training script. https://github.com/yangyixiaof/gitcrawler/tree/master/programprocessor/ngram-train-scripts, 2016.

[23] Y. Yang. N-gram training script tutorial. https://github.com/yangyixiaof/gitcrawler/blob/master/programprocessor/ngram-train-scripts/train-script-tutorial, 2016.

[24] Y. Yang. Ppc program processor. https://github.com/yangyixiaof/gitcrawler/tree/master/programprocessor, 2016.

[25] Y. Yang. Ppc program processor tutorial. https://github.com/yangyixiaof/gitcrawler/blob/master/README.md, 2016.

[26] Y. Yang. The program which puts the model into the aerospike database. https://github.com/yangyixiaof/ModelIntoAeroSpike/tree/master/PutModelIntoAeroSpike, 2016.

[27] Y. Yang. The tutorial of the program which puts the model into the aerospike database. https://github.com/yangyixiaof/ModelIntoAeroSpike/blob/master/README.md, 2016.

[28] Y. Yang. The unzip script. https://github.com/yangyixiaof/gitcrawler/tree/master/gitcrawler/UnzipScript, 2016.