A Language Model for Statements of Software Code

Yixiao Yang, Yu Jiang, Ming Gu, Jiaguang Sun, Jian Gao, Han Liu
School of Software, Tsinghua University, TNLIST, KLISS, Beijing, China

***Abstract*—Building language models for source code enables a large set of improvements on traditional software engineering tasks. One promising application is automatic code completion. State-of-the-art techniques capture code regularities at *token* level with lexical information. Such language models are more suitable for predicting short token sequences, but become less effective with respect to long statement level predictions.**

In this paper, we have proposed PCC to optimize the token-level based language modeling. Specifically, PCC introduced an intermediate representation (IR) for source code, which puts tokens into groups using lexeme and variable relative order. In this way, PCC is able to handle long token sequences, *i.e.*, group sequences, to suggest a complete statement with the precise synthesizer. Further more, PCC employed a fuzzy matching technique which combined genetic and longest common subsequence algorithms to make the prediction more accurate. We have implemented a code completion plugin for Eclipse and evaluated it on open-source Java projects. The results have demonstrated the potential of PCC in generating precise long statement level predictions. In 30%-60% of the cases, it can correctly suggest the complete statement with only six candidates, and 40%-90% of the cases with ten candidates.

***Index Terms*—Code Completion, Language Model, IR**

I. Introduction

Programs, as natural languages, are highly repetitive and predictable [4]. This observation opens a great opportunity to transfer well-designed natural language processing (NLP) techniques to traditional software engineering tasks. Recent years have witnessed a class of researches on building language models for source code [2] [13] [5] [8] [9] [4] [1] [10] [11] [6], which enables a large set of promising applications. In this paper, we focus on automatic code completion, *i.e.*, automatically generate a statement for code suggestion.

Completing one token each time needs users to keep thinking and selecting tokens. Users need to perform many keyboard operations. By comparison, completing a statement is more user-friendly. Previous works have used *n-gram* language model to capture the regularity of source code [4] [1] [10] [11] [6]. The insight is to perform model training using *grams*, *i.e.*, *tokens* of the source code. The previous evaluations have shown the potential of *n-gram* model in predicting *short* code, *i.e.*, short token sequence. However, in terms of *long* predictions, *i.e.*, long complete statement sequence, they become less effective since they only consider token regularity within a specific bound, *i.e.*, *n* as in *n-gram*. Our goal is to overcome such limitation by optimizing the *long* sequence prediction. Unfortunately, to fulfill such a goal in practice is facing the following challenges.

Challenge 1: Long Sequence Regularity. The key of performing long complete statement predictions is to capture long sequence regularity. In the *n-gram* setting, this requires increasing the value of *n*, which may incur a large training overhead and make the model less predictive.

Challenge 2: Large Prediction Space. Furthermore, predicting long sequences leads to exploring on a large prediction space, *i.e.*, all possible subsequent code. The exploration often amounts to something like trying all feasible method calls of a given type, which is quite expensive.

Challenge 3: Complete Statement Synthesize. Finally, synthesizing a complete statement leads to realtime compiling to check whether the suggested statement is legal or not, which is quite time consumption and make the suggestion less effective.

We have proposed PCC in this paper to address the aforementioned challenges. The first ingredient of PCC is an intermediate representation of source code. The intuition of the IR is to group tokens together and capture regularity at group level. In this way, we can use groups to cover more tokens than techniques in the literature. Moreover, PCC introduces a fuzzy matching algorithm to optimize the exploration on large prediction sequence. Specifically, the contexts used to predict are not exactly same as the code contained in training data, but the patterns may be similar. Through searching similar but not accurate matching contexts, we could migrate potential patterns for synthesis. Then, we develop a precise synthesizer to suggest the complete statement efficiently, which adopts realtime compiling techniques to select context-sensitive variables and recombines tokens into a legale complete statement.

For evaluation on open-source Java projects, in 30%-60% of the cases, it can correctly suggest the complete statement with only six candidates, and 40%-90% of the cases with ten candidates, and yield the precision improvement from 34.62% to 41.76% over the n-gram approach. The results demonstrated the potential of PCC in predicting precise statements.

II. Related Work

The statistical n-gram language model has been widely used in capturing patterns in source code. Hindle et al. [4] used n-gram model on lexical tokens to suggest the next token. In SLAMC [6], they enhanced n-gram by associating code tokens with roles, data types, and topics. In cacheca [11], they

Yu Jiang is the corresponding author, School of Software, Tsinghua University (email: jy1989@mail.tsinghua.edu.cn) improved n-gram with caching for recently seen tokens in local files to improve next-token suggestion accuracy. Allamanis et al. [1] and Raychev et al. [10] captured common sequences of API calls with per-object n-grams to predict next call.

Decision tree learning was applied to code suggestion, based on which, Raychev et al. presented an AST-based hybrid language model for source code [8]. Raychev et al.[9] abstracted the code into DSL and kept sampling and validating on a special kind of DSL until the good code suggestion was obtained. Nguyen et al.[5] tried to train code on graphs, and combined Naive-Bayes model and n-gram model to suggest API usages. Recently, deep learning techniques were applied to code suggestion [13] [2]. They found that recurrent neural networks significantly outperform n-grams for doing code suggestion. Given the amount of unstructured code available, state-of-the-art approaches such as recurrent neural networks can outperform existing code suggestion solutions.

Previous works focused on suggesting short token sequence which often represents a method name or a field name. Our work makes a step forward to predict a long statement level sequence which represents a complete statement. We work at the statement level, rather than the lexical level or API level.

III. PCC Approach

The overall architecture of PCC is presented in Figure 1. First, for both training or prediction, the source code would be parsed into our predefined IR, which is a sequence of tokens. In previous works, the token was defined by splitting the source code by white space or punctuation. In the token of our IR, there are many differences. For example, variable names are replaced by some symbols and lexical symbols such as ':' or '}' are eliminated. Then, the *n-gram trainer* trains the IR into n-gram model for statement prediction.

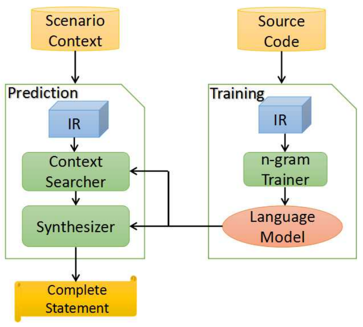


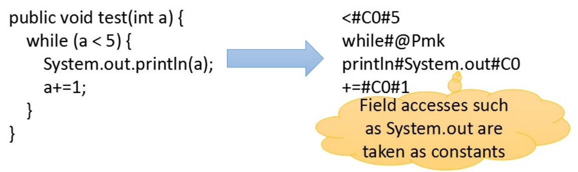
Fig. 1. PCC architecture

For predicting code, the source code prior to the position where users invoke the code completion is taken as the context. The context will be translated to IR. In previous works using n-gram model for prediction, the last few tokens of IR of the context will be used to predict the next token, and the information before the last few tokens is lost. If a value never seen before is observed, the whole probability may be invalid instantly, making it hard for complete statement prediction. For example in 3-gram model, $P(abcd)$ = $P(a|bc)$*$P(b|cd)$*$P(c|d)$*$P(d)$, if c is not observed in training data, the whole probability can not be computed because the values of $P(a|bc)$, $P(b|cd)$ and $P(c|d)$ are unknown. A compromise approach to compute $P(abcd)$ is that $P(abcd)$ = $P(a|b)$*$P(b)$. Information of d is lost. However, the training data may contain bed or bhd instead of bcd, P(abcd) could be computed by $P(abcd) \approx P(abed)$=$P(a|be)$*$P(b|ed)$*$P(e|d)$*$P(d)$ or $P(abcd) \approx P(abhd)$=$P(a|bh)$*$P(b|hd)$*$P(h|d)$*$P(d)$. We design the *context searcher* to find substitutable contexts when some tokens in the origin context are not observed in training data, as in the above example that bcd could be substituted by bed or bhd. The *context searcher* is a long term memory link to bond shattered token snippets. With the *context searcher*, n-gram model could address the challenge of using long contexts to predict next tokens for statement completion.

Finally, when *context searcher* infers n tokens v_1 to v_n from context h, for each $v_{i \in \{1..n\}}$, *synthesizer* judges whether v_i could form a statement without compilation errors. If so, *synthesizer* pushes the generated statement onto the result set, if not, *synthesizer* replaces the context h with $v_i h$, searches from $v_i h$ for each next token $z_{j \in \{1..m\}}$, and judges whether $z_j v_i$ could form a statement or not. Then *synthesizer* replaces the context $v_i h$ with $z_j v_i h$ and does similar searches, and a threshold is set to limit the traversal steps.

A. Intermediate Representation

Intermediate representation(IR) is generated by traversing the AST of source code in post-order. Our IR generation algorithm is applied to the AST generated by Eclipse JDT [12]. A simple example is presented in Figure 2. In Eclipse JDT AST, each node has a type which corresponds to the syntactic information of the node in the AST. There are over 50 types for nodes in Eclipse JDT AST such as $Variable$, $WhileStatement$, $MethodInvocation$. We implement the $GetNodeIdentifier(node)$ function to return the type of the node except for three types of nodes: $Variable$, $Constant$ and $MethodInvocation$. For type $MethodInvocation$, the function should return the name of the invoked method. For type $Constant$, the function should return the concrete value of that constant. For type $Variable$, the function should return the symbol for the variable.



```
public void test(int a) {
   while (a < 5) {
      System.out.println(a);
      a+=1;
   }
}
```

```
<#C0#5
while#@Pmk
println#System.out#C0
+=#C0#1
```

Fig. 2. An example of translating Java source code to IR

Although we can select the suitable variable by type checking and compiling during the code completion stage, we cannot directly give every variable name an unified symbol. Consider the following case. If we decide to choose a variable of a specific type, but there are more than one variable of that type in the context, we can not decide which variable to choose. We observe that if a variable is recently declared, there is a high probability that the variable will be used in the following code. In another word, whether this variable is recently declared could help us classify and choose variables. Based on the observation, when translating a variable var to IR, at each position the variable is used, we give that variable a number which shows how recently the variable is declared compared to other variables with their types declared as same as the type of var. The more recently the variable is declared, the smaller the number is. The minimum of that number is 0. We give every variable symbol a prefix C to denote that this is a symbol for a variable. Figure 3 shows the example of variable translations.

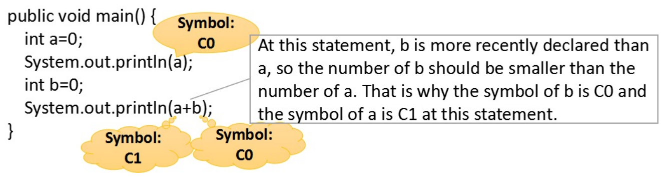

public void main() {
int a=0;
System.out.println(a);
int b=0;
System.out.println(a+b);
}
Symbol: C0
At this statement, b is more recently declared than a, so the number of b should be smaller than the number of a. That is why the symbol of b is C0 and the symbol of a is C1 at this statement.
Symbol: C1
Symbol: C0

Fig. 3. Example of variable translation

Algorithm 1 generates the IR token using post-order traversal. If we are generating the token $token$ for a node $node$, we initialize $token$ to the string gotten by invoking $GetNodeIdentifier$. We iterate child nodes of $node$ from left to right. If the encountered child node represents a variable or a constant, append the separator # and the IR token of that child to the end of $token$. If the encountered child node does not represent a constant or a variable, we think the node is complex and the token of that complex node should be stand-alone without being grouped into any other tokens. So we append the separator # and the string $@Pmk$ to the end of $token$. $@Pmk$ can be thought as a placeholder for the token of that complex AST node. Remember that we use post-order traversal to traverse the AST, the tokens of child nodes are generated before the tokens of parent nodes. Therefore, $@Pmk$ must refer to a previously generated token. The rules about how to identify the token to which $@Pmk$ refers are as follows. For convenience, if a token contains $@Pmk$, we name the token $token\_with\_ref$. At the beginning, we mark all tokens before $token\_with\_ref$ as $un\text{-}referred$. We iterate each $@Pmk$ in $token\_with\_ref$ from right to left. For each encountered $@Pmk$, we find the nearest previous $un\text{-}referred$ token and mark the found token as $referred$. The found token is the one to which the encountered $@Pmk$ refers. Then the $token\_with\_ref$ and all referred tokens must be taken as one token conceptually.

B. Model Training

We only make intra-procedural analysis, so each method in Java files will be parsed into IR which is a token sequence. The source code in Java files in the training data will be parsed into many token sequences where each token sequence corresponds to a Java method. All token sequences will be trained into n-gram model. The model then serves as a server

Algorithm 1: $GenerateIRToken(node)$

```
Input: $node$
Output: $token$
  $token \leftarrow GetNodeIdentifier(node)$
  if $IsVariable(node)$ || $IsConstant(node)$ then
    $return\ token$
  end if
  $children \leftarrow GetChildrenFromLeftToRight(node)$
  for $child : children$ do
    if $IsVariable(child)$ || $IsConstant(child)$ then
      $token \leftarrow token + ``\#'' + GenerateIRToken(child)$
    else
      $token \leftarrow token + ``\#'' + ``@Pmk''$
    end if
  end for
  return $token$
```

to provide the service of searching (inferring) next tokens and the corresponding conditional probabilities given the token sequence with the length not exceeding n-1. The length of a token sequence means the number of tokens in the sequence.

C. Context Searcher

The source code prior to the position waiting to be code completed is taken as the context. In order to obtain the ability to handle unseen tokens, $context\ searcher$ is to find similar contexts according to given context. Algorithm 2 shows the details. In Algorithm 2, for each token in the context, we search for all token sequences which start with that token. At last, we could get a large number of token sequences (possible contexts). Then, we use the function $SortAndMinimizeContexts$ to take similarities, probabilities and lengths of token sequences into a comprehensive consideration to help select suitable token sequences (contexts).

Algorithm 2: $SearchForContexts(given\_context)$

```
Input: $given\_context$ $(must\ be\ a\ token\ list)$
Output: $contexts$
  $contexts \leftarrow \emptyset$
  for $token : given\_context$ do
    $first\_token \leftarrow token$
    $list\_set \leftarrow \{[first\_token]\}$
    $depth \leftarrow 0$
    $max\_depth \leftarrow given\_context.length() * 1.5$
    while $depth < max\_depth$ do
      $new\_list\_set \leftarrow \emptyset$
      for $one\_list : list\_set$ do
        $next\_token\_set \leftarrow InferNextTokens(one\_list)$
        for $next\_token : next\_token\_set$ do
          $new\_list \leftarrow one\_list + next\_token$
          $contexts = contexts \cup new\_list$
          $new\_list\_set = new\_list\_set \cup new\_list$
        end for
      end for
      $list\_set \leftarrow new\_list\_set$
      $depth{++}$
    end while
  end for
  $contexts \leftarrow SortAndMinimizeContexts(contexts)$
  return $contexts$
```

Function $SortAndMinimizeContexts$ uses the longest common subsequence algorithm (LCS) to compute the length of common subsequences between the inferred contexts and the original context. The longer the common subsequences, the higher the priority. If the priorities are same for two inferred contexts, the context with smaller length wins. If two contexts still cannot tell the difference using two metrics above, the context with the higher probability in n-gram model wins. Top ranked sequences will be retained.

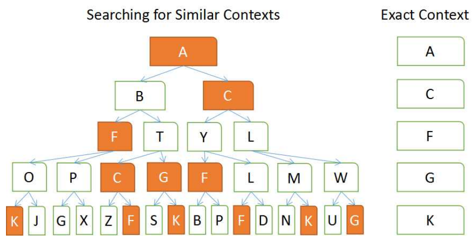

Searching for Similar Contexts
Exact Context
A
B C
F T Y L
O P C G F L M W
K J G X Z F S K B P F D N K U G
A
C
F
G
K

Fig. 4. Example of searching contexts

Figure 4 gives an intuitive graphical representation about this algorithm. For simplicity, complex IR tokens are replaced by English alphabets. Assume that the IR of the context are A, C, F, G, and K. In Figure 4, the algorithm starts at A, keeps inferring next tokens of A to form multiple token sequences which is a path from the root to the leaves. Then, we calculate the LCS between the generated sequence and the original context (A, C, F, G, K). The common subsequences has been marked orange in Figure 4.

D. Synthesizer

Every context found by *context searcher* is fed into a *synthersizer*. Given a context, the *synthesizer* uses the Algorithm 3 to predict all possible token sequences and integrate those token sequences into statements. Algorithm 3 is based on width-first-search. The operator $+$ means concatenation. If a statement could be generated from a sequence successfully, the statement would be appended to final results and we stop exploring that sequence instantly. If not, we must keep predicting next tokens from that sequence to form new sequences to generate statements. The threshold $maxTry$ is set to avoid space explosions.

The key component in Algorithm 3 is $GenerateStatement$ which is responsible for generating statements from a token sequence. It is implemented in Algorithm 4. The whole mechanism is similar to the mechanism JVM uses to execute its byte code. We use a stack to store the generated code. We iterate each token and push the generated code onto stack. A token contains the information of the corresponding AST node and the child nodes of that node. These modules are separated by the separator $\#$. For a token, the function $GetInfoFromRightToLeft$ extracts the modules which are separated by $\#$ from right to left. The uncertain modules are the symbol of variables and the placeholder $@Pmk$. Remember that the $@Pmk$ in a token refers to the *un-referred* nearest previous token. The code of previous tokens

Algorithm 3: $Synthesize(context)$

```
Input: $context$
Output: $statements$
  $statements \leftarrow \emptyset$
  $max\_try \leftarrow 200$
  $try \leftarrow 0$
  $context\_set \leftarrow \{EmptySequence\}$
  while $try < max\_try$ do
    $new\_set \leftarrow \emptyset$
    for $one : context\_set$ do
      $next\_tokens \leftarrow InferNextTokens(context + one)$
      for $token : next\_tokens$ do
        $new\_one \leftarrow one + token$
        $result = GenerateStatement(new\_one)$
        if $result == null$ then
          $new\_set = new\_set \cup new\_one$
        else
          $AppendResult(result)$
        end if
        $try$++
      end for
    end for
    $context\_set \leftarrow new\_set$
  end while
  return $statements$
```

had been pushed onto stack one by one. Therefore, the $@Pmk$ refers to the top of the stack. For each encountered $@Pmk$, we pop the top of stack and replace $@Pmk$ with the popped content. Function $SelectVariables$ handles symbols of variables such as C0, C1 or C2. We take $C0$ as an example to explain the mechanism. Remember that $C0$ represents the most recently declared variable for a type. But in IR, the symbol C0 contains no information about which type this symbol represents, so for each type in the context, the most recently declared variable should be selected. There would be multiple variables selected at the same time. After determining all modules, the function $CombineTogether$ undertakes the tasks of integrating modules together into Java code, checking and compiling the generated code.

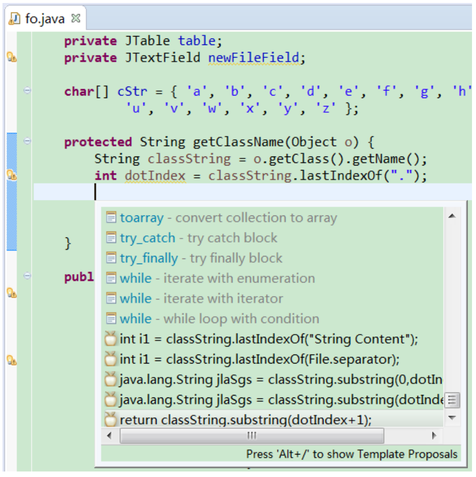
Algorithm 4: $GenerateStatement(material)$

```
Input: $material$
Output: $statement$
  $stack \leftarrow$ $new\ Stack<String>()$
  for $token : material$ do
    $executed\_parts \leftarrow$ $new\ LinkedList<String>()$
    for $part : GetInfoFromRightToLeft(token)$ do
      if $IsVarableSymbol(part)$ then
        $executed\_parts.add(SelectVariables(part))$
      else if $IsPmk(part)$ then
        $executed\_parts.add(stack.pop())$
      else
        $executed\_parts.add(part)$
      end if
    end for
    $part\_stmt \leftarrow CombineTogether(executed\_parts)$
    $stack.push(part\_stmt)$
  end for
  return $stack.pop()$
```

Function $CombineTogether$ integrates modules of a token together according to the syntactic information of that token. Remember that the first module of a token is the syntactic information gotten by invoking $GetNodeIdentifier$. That syntactic information decides how to generate the Java code. For example, for the token $*\#N\#N$, the syntactic information shows that this token should be an infix expression and the operator is $*$, so the generated code is $N * N$. For the token $N\#N\#m1()$, the generated code could be $m1(N, N)$ or $N.m1(N)$. There are over 50 kinds of the syntactic information and each kind corresponds to its own implementation about how to integrate modules together. Function $CombineTogether$ also contains a type checking system implemented by ourselves. There are three kinds of checking. The first kind is to check whether the variable is consistent with its involved arithmetic operator. For example, given an IR token $>\#C0\#2$, variable C0 must be of type int, float, double, long or short. The second kind is to check method specifications. For example, given an IR token $subString\#C0\#C1$, our system checks whether $subString$ can be found in local context. If $subString$ is a method declared in local file and C0, C1 are consistent with parameter types declared by method $subString$, the check is passed and the form of the generated statement will be $subString(C0, C1)$. If the check is not passed, our system takes $subString$ as the member function of C0 and the form of the generated statement will be $C0.subString(C1)$. Then our system checks C0 must be the variable that contains a method named $subString$, C1 must be consistent with the corresponding parameter type. The third kind of checking is to check whether the variable can be casted to the specific type. Our type checking system cannot ensure the completeness and soundness. In order to ensure that there are no compilation errors in final results, we furtively append the statements we generated to the tail of the Java file being code completed, if the whole Java file can be compiled without errors, the statement will be appended to final results. When generating the code for variable declaration such as "$Type\ t = new\ ...$", $CombineTogether$ specifically checks whether the declared type is consistent with the type of the right operand of the assignment operator. If not, $CombineTogether$ will replace the declared type with the actual type of the right operand.

IV. Implementation

The implementation consists of 28555 lines of codes. We implement our system as an Eclipse plug-in named PCC which does not influence the original functionality of code completion in Eclipse. We train 8-gram model and store the model in AeroSpike [3] distributed database. The Antlr4 [7] library is used to parse the special IR. To use the plugin, Users just need to press the hot key to invoke the code completion. Figure 5 shows the screenshots of the running Eclipse with PCC installed. The proposals prefixed by the apple icon are generated by PCC. The lower the position, the higher the priority. The tool PCC [14] is public on GitHub.



```
fo.java
    private JTable table;
    private JTextField newFileField;

    char[] cStr = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'
            'u', 'v', 'w', 'x', 'y', 'z' };

    protected String getClassName(Object o) {
        String classString = o.getClass().getName();
        int dotIndex = classString.lastIndexOf(".");
        |

    }

    publ
```

```
toarray - convert collection to array
try_catch - try catch block
try_finally - try finally block
while - iterate with enumeration
while - iterate with iterator
while - while loop with condition
int i1 = classString.lastIndexOf("String Content");
int i1 = classString.lastIndexOf(File.separator);
java.lang.String jlaSgs = classString.substring(0,dotIn
java.lang.String jlaSgs = classString.substring(dotInd
return classString.substring(dotIndex+1);
Press 'Alt+/' to show Template Proposals
```

Fig. 5. Screenshot of code completing

V. Experiment Results

Experimental Setup. We have conducted the experiments to evaluate the accuracy of code completion at statement level. We pick the *n-gram* based technique [4] as the comparison baseline. All the experiments were performed on a laptop with Intel i7 2.4GHZ processor and 16GB memory. We have collected 500 open-source Java projects from GitHub, which contain 2,465 Java files and 38,183 lines of code in total. The data is available at [15], containing both training data (2,415 files) and testing data (52 files).

Evaluation Process. The code completion is performed at statement level. Specifically, we have commented out each statement in test files and to check whether PCC and *n-gram* model could accurately complete the statement. An accurate completion must match the original code exactly. For PCC, we invoke the code completion engine described in the last section to complete a statement. For *n-gram*, we iteratively predict tokens and record all the possibilities. Then we use the predicted tokens to form a set of sequences and check whether any one of them can match the original code.

Results. The results are public at [15] and shown in Table I. In Table I, $\#Stmt$ is the number of statements in the test file. $TopK$ means the completion matches the right code in first K predictions. Since n-gram model is strongly dependent on plain text, if all variable names in local context are unseen in training data, n-gram model cannot predict the right variable. That is why n-gram model gets 0% accuracy in $CookieCounter.java$. Consider $Top1$ accuracy, in 90% cases, PCC improves the accuracy form 3% to 60%. The average improvement is 22.26%. In the remaining case $DeleteFrame$, PCC gets lower accuracy than n-gram model. The reason is that the context and the predicted statement exist in training data. If the context and the predicted statement are both exactly matched in training data, the n-gram model performs well. Meanwhile PCC predicts many possible statements including the expected statement but the priorities of those statements are slightly wrong. This minor defect in priorities of statements could be eliminated when considering $TopN$ accuracy where

TABLE I
ACCURACY OF CODE COMPLETION

Test File	#Stmt	Top1		Top3		Top6		Top10	
		N-gram	PCC	N-gram	PCC	N-gram	PCC	N-gram	PCC
ArrayMinValue	47	16.7%	29.8%	21.3%	48.9%	25.5%	55.3%	25.5%	57.4%
BallPanel	33	6.1%	9.1%	6.1%	15.6%	6.1%	18.2%	9.1%	21.2%
ClassInfo	35	8.6%	40%	14.3%	71.4%	14.3%	74.3%	14.3%	80%
CookieCounter	21	0%	57.1%	0%	80.9%	0%	85.7%	0%	95.2%
DeleteFrame	37	29.7%	18.9%	35.1%	45.9%	35.1%	51.4%	35.1%	59.5%
DeleteUtil	38	31.6%	34.2%	34.2%	60.5%	34.2%	63.2%	34.2%	63.2%
DrawSquareFrame	10	10%	70%	10%	80%	10%	80%	10%	90%
Foo	52	7.7%	28.8%	11.5%	57.7%	11.5%	57.7%	13.5%	65.4%
FullScreenFrame	23	30.4%	39.1%	47.8%	52.2%	47.8%	65.2%	47.8%	65.2%
JDBCConnCommit	60	11.7%	15%	23.3%	36.7%	26.7%	38.3%	31.7%	41.7%

N is greater than 2. Consider $Top3$ accuracy, in all cases, PCC improves the accuracy form 4.4% to 80.9%. The average improvement is 34.62%. Consider $Top6$ accuracy, in all cases, PCC improves the accuracy form 11.6% to 85.7%. The average improvement is 37.81%. Consider $Top10$ accuracy, in all cases, PCC improves the accuracy form 10% to 95.2%. The average improvement is 41.76%.

The huge search space, the handling of variable names and the elimination of trivial tokens contribute to the improvement. For example, if we want to predict code based on context "$A(B());$", the token sequence of our IR for this code is "$B\ A\#@Pmk$" which contains only 2 tokens. However, the token sequence in n-gram model for this code is "$A\ (\ B\ (\ )\ )\ ;$" which contains 7 tokens. If we use last two tokens to predict code, in n-gram model, we could only use two tokens: ")" and ";" which make little sense. The two tokens: "B" and "$A\#@Pmk$" contain more information than two tokens: ")" and ";" comparatively. Our $context\ searcher$ also contributes to the accuracy improvement through finding patterns from other code. For example, if we predict code from "$Unseen[]\ unsees\ =\ ...;$", every word in the context is unseen before. The $context\ searcher$ finds the similar context "$File[]\ files\ =\ ...;$" and its following statement "$for\ (File\ file\ :\ files)\ ...$". The $synthesizer$ generates the right statement: "$for\ (Unseen\ unsee\ :\ unsees)\ ...$" by selecting and generating suitable variables. Some statements in our experiments are generated in this way. Although our technology has done a lot of searching, the speed of our system is fast, the time consumed to show up the proposals are less than 1 second in all positions waiting to be code completed.

VI. Threats To Validity

The right statements completed in our experiment are mainly JDK APIs, Servlet APIs, common if-judgements and for/while-loops. For other data sets such as programs full of user-defined functions, the result might be different. Our solution to the problem of variable selection does not take the semantics or the topics of variables into consideration and this may produce false positive code which is compilable but semantically wrong. Our approach is based on the linear model with short-term memory. For programs full of branches and complex code relations, our approach is hard to discover the co-relationships of those programs.

VII. Conclusions

In this paper, we proposed PCC to optimize the token-level based language modeling for the statement level code completion. We introduced an intermediate representation (IR) for source code to handle long token sequences, to suggest a complete statement with the precise synthesizer. We also provide a simple solution to problems of variable selection. In the future, we will adopt the graph based model or deep learning to further improve the accuracy of variable selection. Program slicing techniques will also be adopted to analyze long contexts.

References

[1] M. Allamanis and C. A. Sutton. Mining source code repositories at massive scale using language modeling. In *MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216, 2013.

[2] H. K. Dam, T. Tran, and T. Pham. A deep language model for software code. 2016.

[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 *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 *ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 858–868, 2015.

[6] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 532–542, 2013.

[7] T. Parr. Antlr4. http://www.antlr.org/, 2017.

[8] V. Raychev, P. Bielik, and M. T. Vechev. Probabilistic model for code with decision trees. In *OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 731–747, 2016.

[9] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause. Learning programs from noisy data. In *POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016.

[10] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 44, 2014.

[11] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *The ACM Sigsoft International Symposium*, pages 269–280, 2014.

[12] venukb. Eclipse jdt. http://www.eclipse.org/jdt/, 2006.

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

[14] Y. Yang. Pcc. https://github.com/yangyixiaof/CodeCompletionPlugin.

[15] Y. Yang. Experiment data. https://github.com/yangyixiaof/gitcrawler/tree/master/programprocessor/experiment-dataset, 2017.