Statistical API Completion Based on Code Relevance Mining

Chengpeng Wang
Key Laboratory for Information System Security, TNList
School of Software, Tsinghua University
Beijing, China
wangcp16@mails.tsinghua.edu.cn

Yixiao Yang
Key Laboratory for Information System Security, TNList
School of Software, Tsinghua University
Beijing, China
yangyixiaofirst@163.com

Han Liu
Key Laboratory for Information System Security, TNList
School of Software, Tsinghua University
Beijing, China
liuhan0518@163.com

Le Kang
Computer Network Information Center
Chinese Academy of Sciences
Beijing, China
247612299@qq.com

***Abstract*—While Application Programming Interface (API) enables an easy and flexible software development process, selecting a best-fit API is often non-straightforward in practice due to misunderstanding on the API specification or a complex programming context *etc.*. Consequently, the API selection has always been time-consuming and error-prone. In recent years, API recommendation systems have been introduced to help developers choose an API automatically, *e.g.*, `Eclipse` and `IntelliJ` can generate an internal or user-defined API on the fly. Other research leveraged language models to capture the regularity in API usage and further guide the completion of APIs. While existing approaches provided a general support for API usage, they suffer from the lack of semantic awareness (*e.g.*, `Eclipse`) and code relevance (*e.g.*, language model based methods). To overcome these limitations, we proposed CRMAC in this paper. The key insight of CRMAC is a combination of a cache language model which learns code regularity from both open-source projects and local projects, as well as a *relevance mining* engine that identifies similar code to enable a weighted language model training. In our empirical evaluation, CRMAC overwhelmed n-gram approaches, with an improvement of 5.28% in terms of top 10 accuracy. Moreover, over 79% APIs were correctly predicted in the top 10 guesses of CRMAC.**

***Index Terms*—Code completion, n-gram model, code relevance mining, interaction data**

I. Introduction

Code completion assistant is useful for developers in the development of software. If developers do not have code recommendation plugins in the IDE, they have to turn to Java documentation to search the APIs they need. It is apparent that this process costs developers much time and it is really unnecessary. Hence, it is critical to develop a useful API completion tool. However, it is notoriously challenging to predict the API accurately since the information obtained from the code is limited. In particular, we can only get the type of the object in Java code and find the possible APIs by looking up the Java documentation, but in many scenarios, we still need the methods defined in previous code. Moreover, code completion is not robust in some cases because of the absence of current context in the code corpus, which hinders the generation of the correct API usage. Hence, it is vital to delve into this topic and design a powerful tool.

The code in repositories is an important source for mining, and it is a form of interaction data. It stores the programming habits of developers, and can facilitate the coding process of programmers. In practice, several natural language models are applied in the code completion because programming languages share some same features as natural languages [1]. Particularly, they are both written in a specific grammar and repetitive locally. Recent years has witnessed a huge progress on building language models for code prediction [2] [3] [4] [5]. For example, [6] [3] [7] show how to sum up the API context and use key features to decide which API to suggest and [8] [9] [10] split code into tokens and train models such as n-gram and decision-tree to predict a word. Despite wide applications, these tools are not robust enough in some circumstances. Moreover, many of these studies are only based on the models trained in the code corpus, and pay little attention to the projects in development. Some researchers have also proposed a cache model to extract the information from current project [11]. They split the traditional n-gram model into two parts and assign weights to each of them in order to overcome the difficulty mentioned above. One drawback of this model is that it cannot capture the difference between files. Note that n-grams in different files have distinct importance during the model training, we can potentially extend cache model in a more general and powerful form.

In this paper, we present an API-level code completion tool CRMAC. To our best knowledge, this is the first tool which can handle both fuzzy searching and code relevance mining simultaneously. In our work, we select the n-gram model to model the code and apply traditional operations to the n-gram model training such as Lidstone smoothing and back-off operation. In order to make the tool more robust, an algorithm of fuzzy search is proposed to find the content similar to

the current one. Based on these techniques, we are able to generalize the cache model and design an algorithm of code relevance mining. The experiment results demonstrate that our code relevance mining improves the performance of the tool and increases the $top10$ accuracy by 6% on average.

Contribution. Main contributions are summarized below.

- We implement a tool CRMAC for API-level code completion by modeling the code with n-gram model and combining two n-gram models into a cache model.
- We propose the method of fuzzy searching, which can handle some extreme cases and make the tool more robust.
- The concept of code relevance mining is proposed in this paper, and it can improve the performance of completion.

Paper Organization. The rest of this paper is organized as follows. Section 1 is the introduction of API completion. In section 2, the background of program language models is introduced and the n-gram model is explained in details. The design of CRMAC is illustrated in the section 3, including the architecture of the tool, cache model, fuzzy searching and code relevance mining. Section 4 displays the result of experiment, which demonstrates the performance of fuzzy searching and code relevance mining. Section 5 discusses related works and potential directions in future. Conclusions of this work are summarized in the section 6.

II. BACKGROUND

N-gram model is one of the most frequently used models when we model the natural languages and programming languages [12] [13]. It is a simple but effective way to capture the regularity of the languages. N-gram model is aimed to calculate the probability of the sentences. Based on the definition of conditional probability, the probability can be calculated in the following formula:

$$p(a_1a_2...a_k) = p(a_1) * p(a_2|a_1) * ... * p(a_k|a_1a_2...a_{k-1})$$

In an n-gram model, we have an assumption that

$$p(a_l|a_1a_2...a_{l-1}) = p(a_l|a_{l-n+1}...a_{l-1})$$

Hence, if we choose a tri-gram, we can get the approximate formula of the probability

$$p(a_1a_2...a_k) = p(a_1)*p(a_2|a_1)*p(a_3|a_1a_2)*...*p(a_k|a_{k-2}a_{k-1})$$

For a specific sequence, some terms in the formula are equal to 0, so the total probability is 0. Some operations such as smoothing and back-off operation are applied in the n-gram model. In our work, we use **Lidstone smoothing** which is a universal method of smoothing in the model. Lidstone smoothing is a general form of Laplacian smoothing. Take Lidstone smoothing in a tri-gram as an example. Assume $\mathcal{V}$ is dictionary with the capacity of V, and α is a positive number. Then the probability after smoothing is

$$p(a_m|a_{m-1}a_{m-2}) = \frac{count(a_{m-2}, a_{m-1}, a_m) + \alpha}{\Sigma_{a' \in \mathcal{V}} count(a_{m-2}, a_{m-1}, a') + V\alpha}$$

Back-off set the approximate probability of original sequence to the probability of the suffix of the sequence multipled with a discount parameter. In our work, we only define Lidstone smoothing, and it is enough to obtain a good performance.

We calculate the cross entropy of six Java projects modeled by n-gram models. It shows that trigrams can capture enough information, and higher order n-gram models do not have obvious improvements compared with trigrams, while they suffer from high time complexity when being trained. We choose the trigram model as the kernel model.

III. DESIGN OF CRMAC

The overall structure of CRMAC is presented in Figure 1. For training, each file is compared with the file which is being edited, and a training weight is returned to the trainer. The *n-gram trainer* trains two n-gram models in the code corpus and current project, combine them into a *cache model*.

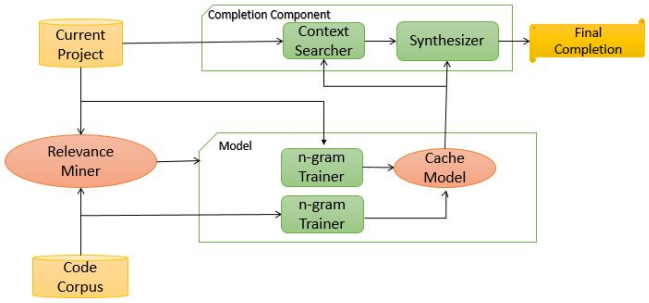

Fig. 1: CRMAC architecture

For predicting, the source code prior to the position users invoke the code completion are taken as the context. Given a context, we design the *context searcher* to find all similar contexts to get a thorough understanding of the given context to improve the accuracy of prediction. Based on the contexts found by the *context searcher*, the *synthesizer* infers a set of APIs by retrieving information from the cache model.

A. Cache Model

Cache model is firstly proposed in [4], and it is similar to a cache-based natural language model [14] in natural language processing. Software code has the property of local repetitiveness, and a code snippet in a project might be written for many times. For example, if a project is developed by a single person, he tends to write FOR statement when he needs loop structure, so the code "for (int i = 0; i < n; i++)" have a high probability to occur in his code.

When training the n-gram model, we have two training sets. One is the external code corpus, and the other is the project which is being developed. In some cases, the code in the current project contains more information we need than the external corpus [15]. In such cases, We can assign a larger weight to the n-grams in the current projects than those in the external corpus to capture this information.

Assume that we have two n-gram models. The model trained in the current project is cache component, and the model

trained in the code corpus is corpus component model. If we need to complete the token in the current file, we can combine these two models in the following way.

$$P(t_i|h, cache) = \lambda * P_{corpus}(t_i|h) + (1 - \lambda) * P_{cache}(t_i|h)$$

The problem is that how to set the value of the parameter λ. If an n-gram occurs many times in current project, the cache component should be assigned much larger weight. Hence, it is obvious that the weight should be relevant to the number of the n-gram occurrence. We can set λ as follows.

$$\lambda = r/(r + H)$$

H is the number of times that prefix occurs in the current project, and r is called concentration factor. When r is fixed to a certain value, the formula satisfies our condition. This is a good design for the weights in the combination of two models.

Algorithm 1 GetProbFromCM

Require: $model_{corpus}$: n-gram model trained in code corpus
$model_{cache}$: n-gram model trained in current project
$context$: content in the current file
r: concentration factor
Ensure: $p(h)$: probability of suffix of current context
1: $h \leftarrow ExtractCompletionPrefix(context)$
2: $H \leftarrow GetOccurenceCount(model_{cache}, h)$
3: $weight1 \leftarrow r/(r + H)$
4: $weight2 \leftarrow H/(r + H)$
5: $prob1 \leftarrow GetProbFromComponent(model_{corpus}, h)$
6: $prob2 \leftarrow GetProbFromComponent(model_{cache}, h)$
7: $prob \leftarrow prob1 * weight1 + prob2 * weight2$
8: **return** $prob$

Algorithm 1 is the key algorithm when we use cache model to calculate the probability. After extracting the context which needs to be completed, the number of context occurrence is computed by the function *GetOccurenceCount*. Then the weights assigned to two components of cache model are calculated. At last, two probabilities are combined in linear form using $weight1$ and $weight2$.

In cache model, concentration factor r is a parameter which is set before the token inference. There is no magical way to get the optimal setting. In most cases, we can obtain a good performance as long as r is not too small. It can be set to the value larger than 10. In the section 4, we compare the accuracy of the model with different values of r.

B. Context Searcher

Language models are weak in handling unseen data which may be similar to the existing data but not exactly the same. Also, as language models suffer from forgetting information from a long time ago, we need to handle long context and unseen data. Fuzzy searching algorithm are applied in this work to find other alternative contexts.

Given a context under which the code to be completed, we find alternative contexts and the code next to contexts found become candidates to be predicted. By computing a small probabilistic model based on the contexts found, We can decide more comprehensively which code following those contexts is the best to be predicted.

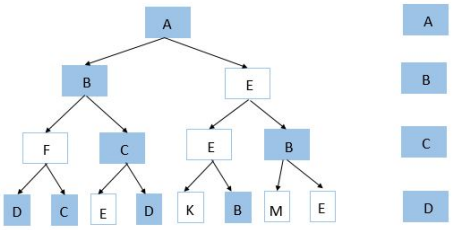

Fig. 2: An example of searching contexts

Algorithm 2 SearchForContexts

Require: $given\_context$ (must be a token list)
Ensure: $contexts \leftarrow \emptyset$
1: **for** $token : given\_context$ **do**
2: $first\_token \leftarrow token$
3: $list\_set \leftarrow \{[first\_token]\}$
4: $depth \leftarrow 0$
5: $max\_depth \leftarrow given\_context.length() * 1.5$
6: **while** $depth < max\_depth$ **do**
7: $new\_list\_set \leftarrow \emptyset$
8: **for** $one\_list : list\_set$ **do**
9: $tset \leftarrow InferNextTokens(one\_list)$
10: **for** $next\_token : tset$ **do**
11: $next\_list \leftarrow one\_list + next\_token$
12: $contexts = contexts \cup new\_list$
13: **end for**
14: **end for**
15: $list\_set \leftarrow new\_list\_set$
16: $depth$++
17: **end while**
18: **end for**
19: $contexts \leftarrow SortAndMinizationContexts(contexts)$
20: **return** $contexts$

We adopt the BFS algorithm to find possible alternative sequences. Algorithm 2 illustrates the details in the context searching. At first, we choose each token in the context and search for all token sequences which start with it. Algorithm invokes the function *InferNextTokens* in line 9 to extend the sequence. Then we get a large number of token sequences which are possible contexts. At last, we use the function *SortAndMinimizeContexts* in line 19 to sort the sequences by their similarity with the context.

Figures 2 displays an example of searching contexts. Give a sequence "ABCD", 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. The tokens in the blue boxes are matched to the tokens in positions of original sequence.

Longest common subsequence algorithm is applied in the function *SortAndMinimizeContexts* to compute the length of

common subsequences between inferred contexts and original contexts. The inferred context with higher ratio of the common subsequence is more likely to be the alternative choice.

C. Relevance Miner

Cache model takes advantages of the difference of importance between n-grams in code corpus and current project. Code corpus and current project are assigned different weights dynamically based on the occurrence of current n-gram, so cache model can be regarded as a rough partition of the training data set. This core insight makes cache model more powerful than the traditional n-gram model.

We can extend cache model to a more general form. The file which is being edited has different relevance to different files in the corpus and current project. It is a natural idea to extract features of these files to compare the similarity between two files. If a file is similar to current file, the n-grams in this file should have larger weight when training n-gram models.

Hence, if we run the algorithm of code relevance mining in the training data set including code corpus and current project, we assign a weight to each n-gram two twice. The weight in cache model is corpus-level. However, in this new model, we calculate the file-level weight. It is obvious that the new model can capture more relevant APIs than the old one.

The remaining problem is that how to extract the feature of a source code file and calculate the similarity between two files. There are various ways to solve it, such as generating the AST trees of code, extracting the features of the AST trees and comparing the features. There are many tools available on line to support such operations. In our work, we segment the function names in the Java file to form a set of substrings. Then we compare two sets obtained from two Java files, and count the common substrings in two sets.

Algorithm 3 FindAPI

Require: $model_{corpus}$: n-gram model trained in corpus
$model_{cache}$: n-gram model trained in current project
r: concentration factor
$text$: code context
Ensure: new_items
1: $token\_list1 \leftarrow getMapValue(model_{corpus}, text)$
2: $token\_list2 \leftarrow getMapValue(model_{cache}, text)$
3: $candidates\_map \leftarrow \emptyset$
4: **for** $token : token\_list1$ **do**
5: $prob \leftarrow GetProbFromCM(model_{corpus}, model_{cache}, r)$
6: $items\_map.put(token, prob)$
7: **end for**
8: **for** $token : token\_list2$ **do**
9: $prob \leftarrow GetProbFromCM(model_{corpus}, model_{cache}, r)$
10: $items\_map.put(token, prob)$
11: **end for**
12: $new\_items \leftarrow ExtractItemsByProb(items\_map)$
13: **return** new_items

D. Synthesizer

When receiving a completion request, the tool scratches the content in current editing environment. Components in cache model are both retrained according to the weight calculated by the *relevance miner*. Then *context searcher* searches for all possible contexts similar to the current context, and every context is fed into a synthesizer. At last, possible APIs are generated by function *FindAPI*.

Algorithm 3 explains each step to search possible APIs when two components of the model have been trained. At first, all possible APIs are retrieved from two components in line 1 and 2. From line 4 to line 10, each probability corresponding to the API is computed and stored in two maps. At last, APIs are sorted based on their corresponding probability.

In Algorithm 4, function *TrainNGram* executes the training phase in the n-gram model from line 4 to line 7. Function *GetProbFromCM* is invoked by *FindAPI* to calculate the probability in the model, and *FindAPI* provides the list of choices sorted by probability.

Algorithm 4 CompleteAPI

Require: $context$: current context
$curfile$: current file
$corpfiles$: the list of Java files in the corpus
$projfiles$: the list of Java files in the current project
r: concentration factor
Ensure: $token\_list$
1: $model_1 \leftarrow \emptyset$
2: $model_2 \leftarrow \emptyset$
3: $token\_list \leftarrow \emptyset$
4: **for** $file : corpfiles$ **do**
5: $weight \leftarrow GetWeight(file, curfile)$
6: $model_1 \leftarrow model_1 \cup TrainNGram(file, weight)$
7: **end for**
8: **for** $file : projfiles$ **do**
9: $weight \leftarrow GetWeight(file, curfile)$
10: $model_2 \leftarrow model_2 \cup TrainNGram(file, weight)$
11: **end for**
12: $contexts \leftarrow SearchForContexts(context)$
13: **for** $text : contexts$ **do**
14: $new\_items \leftarrow FindAPI(model_1, model_2, r, text)$
15: $token\_list \leftarrow token\_list \cup new\_items$
16: **end for**
17: **return** $token\_list$

The input of algorithm 4 includes current code context, current projects, code corpus and concentration factor. Function *CompleteAPI* is the main entry of the tool CRMAC and be invoked when the tool receive a request of API completion.

IV. Experiment Results

CRMAC is implemented in Java and contains modules including *n-gram trainer*, *context searcher*, *relevance miner* and *synthesizer*. *Synthesizer* depends on the other three modules. The kernel model of CRMAC is an array of multiple n-gram

models. If the maximal order of n-gram we need is k, we need to train k n-gram models in order to calculate the probabilities.

Three groups of experiments are performed. In the first group, we set the concentration factor to 0, and cache model becomes traditional n-gram model. CRMAC also uses traditional n-gram model as its kernel. In the second group, we set the concentration factor in the cache model to 10. Concentration factor is set to 100 in the third group of experiments. The details of experiments are following.

Experimental Setup. We have proposed an approach to predict APIs based on the statistical model and code relevance mining. We implemented our approach based on the cache model embedded with relevance miner to predict APIs. In order to demonstrate the effect of cache component in cache model, a group of experiments are also performed by setting the concentration factor to 0. We use traditional n-gram model and cache model respectively to predict APIs in the places where an API occurs. Then we check any one of the APIs in the returned list can match the right API exactly. And then we do the same tasks by using CRMAC. CRMAC, cache model and traditional n-gram model both use trigram model. All the experiments were performed on a laptop with Intel i5-4210M 2.6GHZ processor and 12GB memory. We have collected 6 open-source Java projects to evaluate the performance of CRMAC: Ant, Batik, Cassandra, Log4J, Maven2, Xalan. We use two-fold cross validation, and complete the APIs in 50% files in the testing project. We keep records of $top1$, $top3$, $top5$ and $top10$ accuracy, and calculate the MRR which is the mean value of the reciprocal of the rank. When MRR is close to 1.0, the tool almost achieves the best performance.

Results. The results are shown in Table I, Table II and Table III. We implement experiments in six Java projects. The models are trained in five of them. We use half of files in the remaining project to produce the API recommendations and each project is tested twice.

When the kernel model is n-gram model, on average CRMAC can hit 69.0% of correct APIs in top 10, and the average $top10$ accuracy of traditional n-gram model is 64.8%. When the kernel model is cache model and concentration factor r is 10, both cache model and CRMAC perform much better than traditional n-gram model and CRMAC equipped with traditional n-gram model respectively. Cache component contributes 4.2% improvements in the $top10$ accuracy. Additionally, code relevance mining also improves the performance of the tool. Consider $Top1$ accuracy, CRMAC improves the accuracy from 41.9% to 50.0% in the project Xalan. Consider $Top3$ accuracy, CRMAC improves the accuracy from 50% to 54% in the project Ant. Consider $Top5$ accuracy, CRMAC contributes 10% accuracy improvement in the project Xalan. CRMAC predicts APIs in the project Maven2 very well, and achieve 78% accuracy in $Top10$, and MRRs in this project are 59.0% and 57.3% which are the top 2 MRRs. On average, CRMAC improves the $top3$ accuracy by 4.79% and the $top10$ accuracy by 5.28% when equipped with cache model.

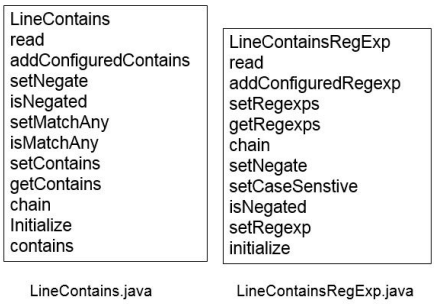

Fig. 3: Lists of function names in two Java files

```
if (line.length() == 1) {
   line = null;
} else {
   line = line.substring(1);
}
```

code snippet 1

```
int ch = line.charAt(linePos);
linePos++;
if (linePos == line.length()) {
   line = null;
}
```

code snippet 2

Fig. 4: Two code snippets

The improvements in the second group of experiments are mainly due to code relevance between editing file and training files. Figure 3 displays an good example. Functions in LineContains.java and LineContainsRegExp.java have the similar names, and two files both contain a same segment of code shown in snippet 1 in Figure 4. When the tool completes the API after "line" in LineContains.java, LineContainsRegExp.java is found as the most similar file to LineContains after the code relevance mining, and it is assigned to a larger training weight. Cache model cannot predict the correct API in this example, and gives "charAt" in the top 1 position. It is due to the occurrence of "charAt" in the code snippet 2 which is in the HeadContains.java. HeadContains.java is less similar to the LineContains.java. Hence "substring" is selected in the code mining. However, if all the training files have the same similarity with the current file, *relevance miner* cant make any contribution to the accuracy. Fortunately, most of files do not have the same similarity with the file which is being edited, so our algorithms perform well.

In the third group of experiments, we set r equal to 100. In this group, the cache component has a larger weight in the cache model compared to the model in the second group. However, the result reveals that changing r from 10 to 100 does not bring the improvement of accuracy. The main reason is that most of n-grams in the current projects do not occur many times, so larger value of r cannot make extra n-grams in the current project selected in the inference phase.

V. RELATED WORK

This paper demonstrates a framework of API completion tool. Some techniques in this work are worthy of further research. In this section, we have a brief discussion about the following related topics.

TABLE I: ACCURACY OF API COMPLETION R=0

Test Project	Top1		Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	39.1%	43.2%	42.3%	47.8%	54.6%	59.4%	62.5%	69.1%	40.2%	45.8%
Ant 2	40.1%	42.2%	44.7%	49.0%	55.9%	60.1%	64.9%	71.1%	41.8%	46.9%
Batik 1	43.2%	47.5%	46.1%	51.0%	60.3%	65.9%	71.0%	72.9%	44.2%	49.0%
Batik 2	40.4%	45.3%	44.2%	49.9%	58.2%	63.3%	68.1%	70.1%	42.1%	47.5%
Cassandra 1	42.1%	45.5%	45.9%	48.8%	54.1%	58.9%	63.9%	69.7%	44.0%	47.2%
Cassandra 2	42.6%	45.1%	46.1%	49.0%	55.2%	59.7%	64.1%	70.2%	44.1%	47.7%
Log4J 1	38.9%	43.4%	43.5%	47.0%	51.0%	56.2%	57.4%	62.1%	40.3%	45.1%
Log4J 2	37.2%	41.3%	42.9%	45.9%	49.5%	54.0%	56.2%	62.0%	39.8%	43.9%
Maven2 1	45.3%	51.2%	50.1%	55.2%	56.1%	62.0%	68.1%	71.0%	47.3%	53.0%
Maven2 2	47.2%	51.7%	52.0%	56.0%	57.9%	63.6%	70.5%	71.2%	49.7%	53.6%
Xalan 1	38.9%	46.0%	43.8%	50.5%	51.7%	62.4%	65.7%	68.1%	40.5%	47.6%
Xalan 2	39.1%	46.8%	43.7%	51.7%	50.1%	63.6%	64.9%	69.9%	41.8%	48.7%

TABLE II: ACCURACY OF API COMPLETION R=10

Test Project	Top1		Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	46.1%	50.8%	50.2%	54.3%	60.5%	64.6%	67.1%	73.2%	48.0%	52.1%
Ant 2	46.1%	50.9%	51.8%	54.7%	62.5%	65.4%	66.8%	74.3%	48.2%	52.3%
Batik 1	45.2%	49.7%	49.8%	53.1%	65.7%	69.3%	73.6%	76.7%	47.3%	51.2%
Batik 2	44.7%	48.9%	48.4%	52.3%	63.1%	68.1%	71.7%	74.2%	46.8%	50.5%
Cassandra 1	46.0%	51.9%	50.7%	56.8%	60.2%	66.4%	69.8%	75.5%	48.1%	54.9%
Cassandra 2	47.0%	54.2%	51.2%	57.5%	62.2%	69.8%	70.8%	77.5%	48.9%	55.7%
Log4J 1	42.9%	45.8%	45.8%	50.1%	54.9%	59.4%	60.1%	68.4%	44.1%	47.4%
Log4J 2	40.1%	44.7%	43.4%	49.8%	51.6%	58.3%	59.5%	67.6%	41.8%	46.9%
Maven2 1	49.3%	55.2%	53.5%	59.6%	58.7%	65.9%	72.6%	77.3%	51.0%	57.3%
Maven2 2	50.2%	57.1%	54.9%	62.3%	59.1%	67.0%	73.4%	78.3%	52.3%	59.0%
Xalan 1	43.6%	50.1%	46.9%	54.7%	56.1%	66.0%	71.4%	74.9%	45.5%	52.8%
Xalan 2	42.9%	49.8%	45.0%	53.9%	54.9%	68.9%	69.8%	72.0%	44.3%	51.9%

TABLE III: ACCURACY OF API COMPLETION R=100

Test Project	Top1		Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	45.8%	47.9%	51.7%	53.2%	58.9%	65.7%	66.9%	74.3%	47.3%	50.2%
Ant 2	45.4%	50.4%	50.1%	55.8%	63.1%	66.2%	68.6%	75.1%	47.1%	52.0%
Batik 1	43.5%	46.7%	48.1%	52.9%	66.2%	70.4%	72.5%	75.9%	45.2%	49.8%
Batik 2	41.4%	46.1%	47.6%	51.2%	65.8%	69.6%	72.9%	74.9%	43.8%	48.2%
Cassandra 1	46.9%	52.7%	51.9%	57.1%	61.7%	67.0%	70.3%	74.2%	48.0%	54.9%
Cassandra 2	47.3%	52.0%	52.4%	56.8%	62.9%	68.9%	71.1%	76.7%	49.8%	53.6%
Log4J 1	40.2%	46.9%	44.9%	51.2%	55.6%	60.1%	62.4%	68.0%	42.1%	48.0%
Log4J 2	40.1%	45.8%	43.7%	50.2%	53.5%	58.9%	61.0%	69.1%	41.5%	47.8%
Maven2 1	49.4%	55.9%	53.7%	59.6%	58.4%	66.4%	72.9%	76.8%	51.0%	57.3%
Maven2 2	50.1%	56.9%	54.5%	61.7%	60.1%	68.2%	73.3%	77.9%	52.3%	58.8%
Xalan 1	42.9%	50.3%	47.3%	55.1%	55.2%	66.7%	71.9%	75.1%	44.1%	52.8%
Xalan 2	41.9%	50.0%	45.8%	54.9%	55.5%	67.5%	70.8%	74.2%	43.6%	52.1%

Heuristic algorithms. Fuzzy search in this work is a heuristic algorithm and can be extended to a more general form. We use LCS algorithm to calculate the similarity between two sequences. Many other methods including genetic algorithms can also give some elegant solutions to this problem.

Code mining is also an important topic in the programming language. In many cases, we can compare two snippets of code by their AST trees. If two trees are almost the same, these two snippets can be regards as the same code. Some parts of code may not have the same token sequences with each other, but they have same semantic information. For instance, the variable names are totally different in two source code files, but an one-to-one map of variables can be established between two files. This problem is not only in the range of the code completion, but also studied in the code clone. [16] [17] [18] [19]

Other language Models. We use the traditional n-gram model in this model. N-gram model cant capture the regularity of long sequence of code, so we need to consider choosing other models. There are many other alternative language models, including skip n-gram model, LSTM and other combinatorial models [20] [21] [22] [23]

VI. CONCLUSIONS

In this paper, we proposed a novel tool CRMAC to complete the APIs in the development of Java project. We modeled the program language via n-gram model and cache model, and utilized context searcher to guarantee that the tool provides the APIs in all possible cases. Moreover, we proposed a straightforward and efficient method of code mining to get the relevant degree between two source files, which made the completion more accurate due to the training weights fed to n-

gram model trainer. Our CRMAC improves the $top3$ accuracy by 4.79 percent points and the $top10$ accuracy by 5.28 percent points. In the future, we will adopt the intermediate representation to implement the expression-level code completion, and try the graph-based model or deep learning model to make full use of the interaction data in the code repositories.

REFERENCES

[1] F.Jacob and R.Tairas, "Code templete inference using language models," in *ACMSE*, 2010, pp. 104:1–104:6.

[2] A.T.Nguyen and T.N.Nguyen, "Graph-based statistical language model for code," in *ICSE*, 2015, pp. 858–868. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.336

[3] V.Raychev, P.Bielik, and M.T.Vechev, "Probabilistic model for code with decision trees," in *OOPSLA*, 2016, pp. 731–747. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984041

[4] Z.Tu, Z.Su, and P.Devanbu, "On the localness of software," in *The ACM Sigsoft International Symposium*, 2014, pp. 269–280.

[5] M.Gabel and Z.Su, "A study of the uniqueness of source code," in *FSE*, 2010, pp. 147–156.

[6] V.Raychev, M.T.Vechev, and E.Yahav, "Code completion with statistical language models," in *PLDI*, 2014, p. 44. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[7] Y.Yang, Y.Jiang, M.Gu, J.Sun, J.Gao, and H.Liu, "A language model for statements of software code," in *ASE*. IEEE Press, 2017, pp. 682–687.

[8] K.Hoa, T.Tran, and T.Pham, "A deep language model for software code," in *arXiv preprint arXiv:1608.02715*.

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

[10] 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*, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[11] C.Franks, Z.Tu, P.Devanbu, and V.Hellendoorn, "Cacheca: A cache language model based code suggestion tool," in *ICSE*, 2015, pp. 705–708.

[12] W. B. C and T. M, "N-gram-based text categorization," vol. 48113, no. 2. Citeseer, 1994, pp. 161–175.

[13] P.F.Brown, P.V.deSouza, R.L.Mercer, V.J.D.Pietra, and J.C.Lai, "Class-based n-gram models of natural language," in *Computational Linguistics, 16(2)*, 1992, pp. 467–479.

[14] R.Kuhn and R.D.Mori, "A cache-based natural language model for speech recognition," in *IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(6)*, 1990, pp. 570–583.

[15] R.Robbes and M.Lanza, "Improving code completion with program history," in *Automated Software Engineering 17(2)*, 2010, pp. 257–277.

[16] J.Gao, X.Yang, Y.Fu, Y.Jiang, and J.Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *ASE*. ACM, 2018, pp. 896–899.

[17] J.Gao, X.Yang, Y.Fu, Y.Jiang, H.Shi, and J.Sun, "Vulseeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation," in *FSE*. ACM, 2018, pp. 803–808.

[18] C.Wang, Y.Jiang, X.Zhao, X.Song, M.Gu, and J.Sun, "Weak-assert: A weakness-oriented assertion recommendation toolkit for program analysis," in *ICSE(Companion Volume)*. IEEE, 2018, pp. 69–72.

[19] H.Liu, Z.Yang, C.Liu, Y.Jiang, W.Zhao, and J.Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 900–903.

[20] S.Andreas, Z.Jing, W.Wen, and A.Victor, "Srilm at sixteen: Update and outlook," in *Proceedings of IEEE Automatic Speech Recognition and Understanding Workshop*, 2011, p. 5.

[21] X.Wang, M.Andrew, and W.Xing, "Topical n-grams: Phrase and topic discovery, with an application to information retrieval," in *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*. IEEE, 2007, pp. 697–702.

[22] P.Bielik, V.Raychev, and M.Vechev, "Phog: Probabilistic model for code," in *ICML*, 2016.

[23] Y.Jiang, H.Zhang, H.Liu, X.Song, W.Hung, M.Gu, and J.Sun, "System reliability calculation based on the run-time analysis of ladder program," in *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 695–698.