2020-128-(x): Manuscript for presentation at IPSJ-SIGPRO, 12 3 2020.

Improving Keyword-based Code Recommendation by Exploiting Context Information

Shu Aochi,[a) Hidehiko Masuhara,[b)

Abstract: Code recommendation provides code fragments that the programmer likely to type in. One of the advanced code recommendation techniques is keyword programming, which can reflect the programmers' intention. Keyword programming lets the user specify keywords and recommends expressions that contain as many of them. Another one is neural code completion, which uses neural networks to recommend likely occurring expressions according to the context (the program text preceding the cursor position). Previous work showed that the accuracy of a keyword programming system is not high enough. One of the reasons is that the existing keyword programming always recommends shorter expressions without using the context information. In this presentation, we improve keyword programming by combining a neural code completion technique. In addition to the occurrence of keyword, the ranking algorithm incorporates the likeliness factor of the code fragment concerning the context. To estimate the likeliness, we utilize a neural network-based sentence generator. Thus, we can achieve a more complicatedly suitable code fragment and generate a candidate list varying along with different contexts. We implemented our proposal for Java called ACKN as an Eclipse plug-in. The implementation is publicly available.

1. Introduction

Code recommendation, also called code completion, is one of the common features in modern programming editors that presents a list of code fragments to the programmer so that he or she can input the desired code by merely choosing one of them. While a typical implementation presents the names of available methods/functions/variables that match the letters already typed in the editor, there are many variations with respect to the lengths of the presented code fragments (from function names to a few lines of code), and with respect to the information used for making recommendations.

The information that a code recommendation takes into consideration can be divided into two kinds: explicit and implicit.

Explicit information stands for straightforward information that comes from inputs of a code recommendation system. And for such kinds of systems, there are three categories of input: an abbreviation, a partial expression and a bunch of keywords.

Traditional code recommendation systems like the default one on Eclipse provide all possible code fragments after the user inputs a prefix of an identifier. The recommendations are normally sorted alphabetically, the user could browse the recommendation list and select the required code fragment.

For example, suppose a user wants to write a code that can read standard input from the console. In Java, the expression would be `new BufferedReader(new InputStreamReader(System.in))`. Instead of typing all strokes, the user can only type the prefix *new Bu* to get the identifier `BufferedReader`, *new InputS* for `InputStreamReader`, *Sy* for `System` and *i* for the last one `in`.

Romain Robbes et al. [1] propose a code recommendation system that also inputs a few characters, but the recommendations are sorted according to the user's programming history.

Moreover, Sangmok Han et al. [2] and Sheng Hu et al. [3] recommend possible code fragments given an abbreviated input. The former uses a Hidden Markov Model to expand the abbreviation to inputs, while the latter uses a Gaussian mixture model.

In addition, Greg little et al. [4] propose an approach that the user types some keywords to search for an expected expression. And after the keyword programming system provides a list of candidate expressions, the user can look through the list and select the expected one.

Unlike the default code recommendation system on Eclipse, the user only needs to trigger the keyword programming system once to get the expected expression. For example, after type keywords **buffered reader in** and run the keyword programming system, the expression `new BufferedReader(InputStreamReader(System.in))` will be shown on the top of the recommendation list.

In contrast, implicit information denotes information that can be exploited from the context. These kinds of systems can be categorized by different statistical models.

For example, TabNine [5] and Marcel Bruch et al. [6] takes the whole context into consideration arranges the recommendations by their probabilities. The previous system cal-

a) shuaochi@prg.is.titech.ac.jp
b) masuhara@is.titech.ac.jp

culates probabilities by a GPT-2 model and Marcel Bruch et al. utilize an algorithm named Best Matching Neighbors(BMN) based on the K-Nearest Neighbors algorithm.

On the other hand, neural networks have developed rapidly in the past 20 years and have shown their strong power in many genres. One of them is the text generation, which aims to generate a likely sentence. Those neural networks text generators are divided by using different neural networks. [7]

In 2003, Bengio et al. [8] first use a standard neural model that extends the n-gram paradigm with neural networks.

Then,a recurrent neural network language model [9] is used in text generation because it can process time-series data.

Moreover, a long short term model(LSTM) [10] is an optimized RNN designed for learning the long term dependency. In contrast to the standard RNN, LSTM has a better performance to learn from more complicated data.

In this paper, we propose a code recommendation system that improves the existing keyword programming system by using neural text generation to concern the context of the user ' s editing file.

Our goal is to build a code recommendation system that:

(1) Suitable to all programmers including a beginner and an expert.

(2) Recommend the expression that satisfies the user's purpose, especially when the expression is complex.

We implement a plug-in on Eclipse named ACKN based on a context-aware keyword programming technique.

2. Background

2.1 Keyword Programming

Keyword programming is a technique that translates a keyword query to an expression. A keyword programming system has four parts, namely input, extraction, generation, and ranking.

The user first inputs a keyword query in the source code. Then the system collects the information of local variables and methods from the source code and generates all possible expressions by following the syntax and typing rules. Finally, the system calculates the scores of those expressions and shows that have higher scores.

2.1.1 Input

The keyword query is similar to the search query of a search engine, which conveys the programmer ' s intention. For example, if a programmer wants to write an expression to display the name of a variable `f` on the monitor, the keyword query would be **print f name**. If the user wants to get the max number between number a and number b, they could use **get maximum between a and b**.

The keyword query is the whole tokens in the line where the cursor position is in the active editor. From the information of a cursor position, the system can know which local variable can be referred and which method can be called in that position.

2.1.2 Extraction

Keyword programming extracts essential information before generation. The information includes:

- all available object class name,
- the name and type of a local variable,
- the name, type, and object class of a field, and
- the name, return type, object class, and argument types of a method.

For example, supposed the system wants to generate the expression `System.out.println(result)`. First, it is necessary to know the information of a class name `System`. And a field `out` that is in the object class *System* and return the type *PrintStream*. Also, a method named `println` where the receiver type is *PrintStream*, the return type is *void* and the argument type is *String*. Finally, a local variable `result` that returns the type *String*.

2.1.3 Generation

After obtaining those elements from a source file, the system generates all possible expressions that obey the syntax and type rules of the programming language. For instance, considering the program

```
1  package example;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.util.ArrayList;
6  import java.util.List;
7
8  public class Example{
9    public static void main(String[] args){
10       String result = ‘‘result’’;
11   }
12
13   public List getLines(BufferedReader src)
       throws IOException{
14       List array = new ArrayList();
15       while(src.ready()){
16           |
17       }
18
19       return null;
20   }
21 }
```

The cursor is in line 16. In this line, we can call the method and field in the class *BufferedReader*, *IOException*, *ArrayList* and *List*. Moreover, we also can refer the variable `src` and `array`. Therefore, a method invocation such as `array.add(src.readLine())` is available, where `add` is one of the methods declared in the class *List*, and `readLine` is a method declared in the class *BufferedReader*.

However, an expression like `System.out.print(result)` would lead to a compile error, because it is not able to refer the variable `result` in the line 10. Another method invocation such as `result.print()` is also forbidden since it does not follow the type rules.

2.1.4 Ranking

The code completion system of Eclipse organizes the recommendation alphabetically. For example, the method `add` shows in front of the method `concat` on the recommendation list.

In contrast, a keyword programming system shows the recommendations in the order of scores calculated by rules that:

- -0.05 for each height of abstract syntax tree,
- +1.0 if the expression contains a token inside the keyword query,
- -0.01 if the token is not in the keyword query, and
- +0.001, when the element is a local variable or a member method.

Token is obtained by splitting an expression with punctuation and upper case letter. For example, the expression `str.toUpperCase()` has 4 tokens, which are `str`, `to`, `upper` and `case`.

We demonstrate the rule by an example of calculating the score for a function `array.add()` given the keyword **add** and **line**.

First, since the height of `array.add()` is two, the initial score is -0.1.

Then, since the token `array` is not in the keyword query, the score becomes -0.11.

Because the token `add` is in the keyword query, the score becomes +0.89.

Finally, since the token array is also a local variable, the final score is +0.891.

2.2 Neural text generation

Word2Vec Mechnism

3. Problem

Although keyword programming can help a user to complete the program conveniently, there is still a problem that the expected expression, especially a complicated one, is not able to be shown as the first candidate.

For example, if a user wants to read the standard input information from console, in Java, most of users would write the expression `new BufferedReader(new InputStreamReader(System.in))`.

When the programmer uses keyword programming to get this expression, after the user types the keywords **reader in** and triggers the system, the first candidate would be `new InputStreamReader(System.in)`, and the 12th candidate is the expected expression `new BufferedReader(new InputStreamReader(System.in))`.

In the previous research, an expression is 90% to be shown on the top of the recommendation list if the keywords use as same tokens as in the expression. For example, given a keyword query **new buffered reader new input stream reader system in**, the expression `new BufferedReader(new InputStreamReader(System.in))` would be the first candidate.

However, like other code recommendation technique, keyword programming aims to shorten programming time. Thus, it would be meaningless if the user types all tokens in the expected expression. Alternatively, a keyword programming user would be more likely to choose relatively fewer substrings in the expected expression as the keyword query such as **reader in**.

By using the scoring function in the section 2.1.4, the score of the expression `new InputStreamReader(System.in)` would be +1.81. On the other hand, the score of the expression `new BufferedReader(new InputStreamReader (System.in))` would be +1.74. Therefore, the former expression is in a higher position in the recommendation list.

The reason is that an expression has a higher score calculated by the scoring function when it is shorter and contains more keywords. Therefore, when the expected expression becomes more complicated and the number of keywords is relatively fewer, it is hard to be shown in a higher position in the recommendation list. In this paper, complicate denotes that the expression has a larger abstract syntax tree or contains many tokens. For instance, the height of `new BufferedReader(new InputStreamReader(System.in))` is 4, and it contains 9 tokens from `new` to `in`.

4. Proposal: Using a NN text generator

Our proposal is to use a neural network text generator to recommend more context-dependent expressions.

The approach is straightforward. For each expression, we calculate not only a score by the orginal scoring function but also the occurance probability. We add these two values to represent the final score of the expression.

For example, if the user is editing a program where the cursor is in line 6, and the keyword query is `read in`:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ReadInput{
  public static void main(String[] args){
      reader in|
  }
}
```

By using the original scoring function, the score of the expression is +1.74.

Subsequently, to calculate the occurance probability of the expression `new BufferedReader(new InputStreamReader(System.in))`, we need 7 steps.

First, we take the token sequence from `import` to `args` as an argument and predict the probability of the token `new` by the neural network model.

Then, append the token `new` to the token sequence and predict the probability of the token `BufferReader`. Repeat this procedure until get the probability of the last token `in`.

At this moment, we have 6 probability numbers for each tokens. Finally, we add these 6 probabilities and use it to represent the occurance probability of the expression.

Thus, in the new scoring function, the final score for the expression `new BufferedReader(new InputStreamReader(System.in))` is +(1.74 + *probability*).

By exploiting the implicit information from the previous codes and considering the context, an expression would have a higher probability value if it is more likely to be the next expression.

These implicit information includes the imported package declaration, the identifier names and the order of previous method invocations. For example, in the preceding program, the program has imported two package *java.io.BufferedReader* and *java.io.InputStreamReader*. Moreover, the class name is *ReadInput*. The user is more likely to write the expression `new BufferedReader(new InputStreamReader(System.in))` than `new InputStreamReader(System.in)`. Therefore, the probability of the former expression is higher than the latter one. In other words, the candidate expression `new BufferedReader(new InputStreamReader(System.in))` could be shown in a higher position.

5. Implementation

Beam Search
Word2Vec
LSTM

6. Evaluation

6.1 Procedure

6.2 Result

6.3 Discussion

7. Future Work

Word2Vec Speed: NN

8. Related Work

9. Conclusion

References

[1] Robbes R, Lanza M. How program history can improve code completion[C]//2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2008: 317-326.

[2] Han S, Wallace D R, Miller R C. Code completion from abbreviated input[C]//2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009: 332-343.

[3] Hu S, Xiao C, Qin J, et al. Autocompletion for Prefix-Abbreviated Input[C]//Proceedings of the 2019 International Conference on Management of Data. 2019: 211-228.

[4] Little G, Miller R C. Keyword programming in Java[J]. Automated Software Engineering, 2009, 16(1): 37.

[5] TabNine: https://tabnine.com/

[6] Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems[C]//Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. 2009: 213-222.

[7] Lu S, Zhu Y, Zhang W, et al. Neural text generation: Past, present and beyond[J]. arXiv preprint arXiv:1803.07133, 2018.

[8] Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model[J]. Journal of machine learning research, 2003, 3(Feb): 1137-1155.

[9] Mikolov T, Karafiát M, Burget L, et al. Recurrent neural network based language model[C]//Eleventh annual conference of the international speech communication association. 2010.

[10] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.