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

Selecting a proper programming environment is essential for a programmer when they start to build a project. A Java programmer is likely to choose Eclipse or IntelliJ, while a C programmer prefers Visual Studio. A principle to judge which environment is appropriate is the efficiency in developing code.

Code recommendation is one of the advanced techniques that assist a programmer to save time by providing a list of possible code fragments. The recommendations would show on the editing monitor after a user type some inputs. Then, the user can view the list and select the desired code fragment instead of writing down the whole code.

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

[a] shuaochi@prg.is.titech.ac.jp
[b] masuhara@is.titech.ac.jp

the whole context into consideration arranges the recommendations by their probabilities. The previous system calculates 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 The cursor is in line 18. 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, a method invocation such as 'System.out.print(result)' leads to a compile error, because it is not able to refer the variable 'result' in line 18. 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.

Likewise, 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.

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 depth 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" belongs to the keyword query, the score becomes +0.89.

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

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.