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 information used for making recommendations.

Information used for making recommendations can roughly be classified into two kinds, namely *explicit* intention and *implicit* context.

Explicit intention is the information that the programmer provides to the system when he or she wants to obtain recommendation. Examples are *prefix letters*, *an abbreviation*, and *keywords*. When the programmer wants to type `InputStreamReader`, he or she can type "`Inp`" or "`ISR`" to a prefix-based or abbreviation-based system, respectively, then the system will generate identifiers that start with `Inp` or that are concatenation of words starting from `I`, `S` and `R`. These kind of recommendation systems can be found many programming editors, for example Eclipse IDE.

Keyword programming, on which we are based on, uses keywords[*1] as the explicit intention [4]. The key idea is, by letting the programmer provide a bit longer explicit intention, to enable the system can recommend longer expressions or statements. For example, when the programmer wants to input

```
new BufferedReader(
    new InputStreamReader(System.in)),
```

he or she can type "`buffered reader in`" so that the system will recommend expressions including the above one. (In the next section, we explain how keyword programming generates recommendations.)

Implicit context is the information available in the code and the past behaviors of the programmer. For example, the recommendation system in Eclipse uses the type information of the expressions around the cursor position, and recommends identifiers (class, variable or method names) that can form an expression with a matching type.

Implicit context can improve the quality of recommendations. For example, Robbes et al. discovered that the type information and code structure are useful to greatly improve a prefix-based recommendation system [1].

Han et al. proposed a method to improve an abbreviation-based recommendation system by using a hidden Markov model (HMM) [2]. They construct a HMM of a program corpus, and use it for recommending expressions that are more likely to appear in the corpus.

Many recommendation systems use implicit context information in combination with knowledge from a corpus. In other words, those systems recommend expressions/identifiers that "programmers who wrote these also wrote." For example, TabNine [5] and Bruch et al.'s work [6] extract a

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

*1 In keyword programmings, the term *keyword* means a query word used for searching, like the one used for web search engines. It should not be confused with keywords (or reserved words) in the syntax of programming languages.

sequence of tokens or a sequence of method calls before the cursor position in the editor, and recommend expressions or identifiers that frequently appear in the expressions that share the same sequence in the corpus.

Those system exploit corpora by using statistical methods in order to cope with large corpora. Bruch et al. use a clustering method, for example. TabNine uses a deep learning method **?** called GPT-2 **?**. Our work also uses a deep learning method for augmenting the keyword programming with the corpus knowledge. We will explain a fundamental mechanism of using a deep learning method for exploiting a corpus in the next section.

This paper proposes a method of improving the keyword programming by exploiting program corpus knowledge. With the support from a neural network based sentence generator, it tries to recommend expressions not just containing the programmer provided keywords, but those more likely appear in the corpus. By dosing so, we aim at making the keyword programming usable for larger expressions.

The rest of the paper is organized as follows. We first introduce how the original keyword programming recommends expressions based on the generate-and-ranking method. We also overview what a neural network can generate sentences with a corpus (Section 2). We then illustrate that the original keyword programming works poorly for recommending larger expressions (Section 3). We present our proposal, which lets the keyword programming recommend expressions that more likely appear in the corpus, based on the probabilities from the sentence generator (Section 4). We implemented the proposal as an Eclipse plug-in called ACKN (Section 5) and evaluated the proposal by performing an experiment of inserting 15 expressions with the original and proposed keyword programming (Section 6).

2. Background

2.1 Keyword Programming

The keyword programming [4] is an expression-level recommendation system based on keywords. It is used when the programmer wants to write an expression (*the desired expression*)[*2] in a partially written program (*the program context*), by typing a set of words (*the keywords*) into the position where it should appear (*the cursor position*). The system then shows an ordered list of expressions (*the recommendations*), one of which will be selected by the programmer and inserted into the cursor position. Below, we explain the mechanism in the four steps, namely reading keywords, extracting context information, expression generation, and ranking.

2.1.1 Keywords

The keywords are set of words separated by a space character, and should be part of the desired expression. Similar to the keyword-based web search engines, the user of the keyword programming needs to select the words that represent the desired expression well.

Even when the programmer does not exactly know the desired expression, it is possible to provide keywords by using the terms at the semantic level. For example, when the programmer wants to display the file name of the variable `f` of the type `File`, providing "`print f name`" can lead to a recommendation `System.out.print(f.getName())` without knowing the exact method name.

2.1.2 Extracting Context Information

As program context, the system extracts information of

- the variables and their types that can be referenced at the cursor position, and
- the types (including method and field signatures) available in the program files.

The information will be used for generating expressions that have valid types in the next step.

2.1.3 Expression Generation

The system then generates all valid expressions and passes them to the next ranking step. A "valid" expression is such an expression that will be safely compiled when it is inserted at the cursor position. Since there can be infinitely many valid expressions, an implementation actually generates only expressions with a limited size, and avoids generating expressions that will have lower scores in the next step.

2.1.4 Ranking

The system calculates scores of the generated expressions in order to show top N recommendations. The scoring function prefers expressions that are more concise and have more keywords. The function is defined in this formula:

$$-0.05N + 1.0K - 0.01D + 0.001L$$

where

- N is the nesting level of the expression (i.e., the height of its abstract syntax tree representation),
- K is the number of keywords that match tokens in the expression,
- D is the number of tokens in the expression that do not match any keyword, and
- L is the number of tokens that reference local variables or method names.

Here, a token is a component of an identifier split by the "camel case." For example, the method name `getName` consists of the two tokens namely `get` and `name`; hence the keyword `name` is considered as appeared therein.

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] The keyword programming system can recommend not only expressions but also one or multiple statements. Here, we only explain the case for expressions for simplicity.

2.2 Neural network text generation

A neural network text generation can be accomplished in 4 steps. We demonstrate this by an example of predicting the next word given a sentence in Shakespeare's style.

First, the system reads the data and create the dictionary of words. The data includes all Shakespeare's works. Moreover, because neural networks can only process with numeric values, it need a way to represent word by numbers, namely word embedding. There are many ways to map words, such as one-hot embedding and word2vec.

Second, the system builds 2 lists that are used as the inputs of the neural networks. One list contains a word sequence, and the other one contains the next word. For example, the sentence *"Over hill, over dale"* can create 3 pairs of data. The first sequence is *over* and the next word is *hill*. The second sequence is *over hill* and the next word is *over*. The last sequence is *over hill over* and the corresponding next word is *dale*. After obtaining the data, we need to map each word to a numeric value by using word embedding.

Third, input the data and train the neural networks with it.

Finally, predict the probability of the next word given a word sequence. For example, given a sentence *ake thee of* then the word *thy* would have the highest probability to be the next word calculated by the neural networks model.

Moreover, if the user wants to generate a sentence, they need to iterate the last procedure after append the word with the highest probability to the end of the previous word sequence. To be more specific, the probability of the next word is calculated by the neural model given the word sequence *ake thee of thy*.

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`:

```
1 import java.io.BufferedReader;
2 import java.io.InputStreamReader;
3
4 public class ReadInput{
5     public static void main(String[] args){
6         reader in|
7     }
8 }
```

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 3 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 classes *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.

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.FileReader;
4
5  public class Read{
6      public static void main(String[] args){
7          String path = "foo.in";
8          reader file|
9      }
10 }
```

Neural network text generation and keyword programming are complementary, that is why we conbine these two technique.

Neural network text generation could distinguish which expression is likely to be used afterwards. For example, if the program imports classes *java.io.BufferedReader*, *java.io.InputStreamReader* and *java.io.FileReader*. Both `new BufferedReader(new InputStreamReader(System.in))` and `new BufferedReader(new FileReader(path))` have a higher probability to be the next expression.

Then keyword programming can represent the user's intention. For example, if they want to read the input from the specified file, the keyword query would be **reader file**. Then the candidate `new BufferedReader(new FileReader(path))` is more likely to be shown in the recommenedation list. On the other hand, if the user wants to read the input from the console, then the keyword query becomes **reader in**. As a result, `new BufferedReader(new InputStreamReader(System.in))` would be shown in the toppest of the recommendation list.

5. Implementation

We build an Eclipse plug-in to implement our idea. The plug-in is written in Java. We named this plug-in as ACKN, which shorts for **A**uto **C**ompletion with **K**eyword programming and **N**eural network text generation.

5.1 Search expression from all generations

It is a search problem to select partial expressions to show on the recommendation list from all possible generations. Although a static programming language like Java has type constraints that can limit the size of generations, it is still unpossible to show all available candidates.

In the previous research, they used dynamic programming and A* algorithm. In ACKN, we use another greedy algorithm called beam search.

Beam search is a heuristic graph search algorithm. It is based on breadth first search with width constraint. To be more specific, in each depth, it first sorts the candidates by a scoring function. Then, instead of all branches, beam search cut the branches with lower score and continue to search on the remaining branches. *bw* stands for the number of remaining branches for each depth.

For example, suppose we want to select expressions with beam search from generations shown in Figure x. Given the keyword query is **read**, the system can calculate the score for each generation. With beam search, if the *bw* is 3, then the beam search algorithm is same as breadth first search. If the *bw* is 2, then the second branch of depth 2 and the third branch of depth 3 would be eliminated. In other words, the expression and would remain as the result.

5.2 Preprocessing the training data

Subsequently, we need 2 steps before learning from the neural networks. One is to reduce some unnecessary information from the training program, and another is to transform the code into numbers.

First, we eliminate the comment of each program and transform all string literal to *"stringliteral"*. Since we want to focus on the information from methods and variables, we ignore these noise.

After this step, we use word2vec to map each token to a vector value.

There are two reasons why we use word2vec.

First, in contrast to traditional embedding approach such as one-hot embedding, word2vec can represent a word in a lower dimension. Thus, it requires less memory to store the data and it is faster to train the neural networks model with those data.

Second, word2vec can make sentence generator recognize synonyms in the context. Different programmers have different naming styles, word2vec can ignore the difference between two similar words to the greatest extend.

For example, in figure x, program A and B are two programs that has a difference in line x. After using word2vec to map each word to a vector, we can measure the similarity of these two word by calculating the cosine similarity between these two vectors. In this case, the similarity between and is . Therefore, these two programs can be treated as similar programs.

We implemented the word2vec by using the gensim package.

Moreover, we use LSTM as the neural networks model

of the neural network text generator. We implement the LSTM networks by using the keras deep learning package.

6. Evaluation

6.1 Procedure

We evaluate ACKN by following five steps.

First, we prepare 15 expressions decided by ourselves.

Then, for each expression, we search on the github and collect 21 programs containing the expression in the former step.

After we get the programs, we add 20 of 21 programs for each to the training dataset of the LSTM model. In order to avoid overfitting, we also add 5000 Java programs from the dataset built to evaluate the implementation in the [11], which are the top active Java GitHub projects on January 22nd 2015.

Subsequently, we create a task for each by using the remaining one program. We erase the code of the expression in that program. and make it to be a program with a hole. And for each expression, we think a keyword query that can describe the expression.

Finally, we run each task on both ACKN and the original keyword programming system. Then, compare the position where the expected expression is in each recommendation list. The recommendation list only show the first 30 results.

6.2 Result

Figure x shows the expected expression and corresponding keyword.

Figure x shows the rank of the expected expression in the result and the first result by using the original keyword programming system. Figure x shows the rank and the first result by using ACKN.

By using the existing keyword programming,11 tasks can show the expected expression on the list and the expected expression are in the top-5 results in 8 tasks. In contrast, by using ACKN, 10 tasks can show on the recommendation list, and 5 tasks are in the top-5 results.

7. Future Work

7.1 Support synonyms in the keyword query

A keyword programming user would use similar words to the substrings of the expected expression in the keyword query. For example, if the user wants to print a file's name on the console, the expression would be `System.out.print(filename)`. Instead of using **print filename** as the keyword query, the user would type **display path**. For a human being, the meaning between two keyword querys are basically identical, whereas it is different for a machine.

One approach is to use word2vec. As we introduced in the section 5.2, variable names such as `filename` and `path` can be recognized as synonym when the surrounding codes are similar.

However, the word `display` and `print` can not be counted as similar with this approach. Because in the programming world, similarity stands for similar function, not similar meaning in the natural language. For instance, `print` and `println` can be considered as similar tokens. Because they are often written after the code fragment `System.out`.

7.2 Shorten completion time

In ACKN, we have to calculate the probability for each generation. Thus, it always costs few minutes to get the result and show it on the monitor, which are not acceptable for a code recommendation system.

7.3 Using different neural networks

Except LSTM, there are also many neural networks that can be used in text generation. For example, the gated recurrent unit(GRU), generative pre-training(GPT), or generative adversarial nets(GANs). In addition, the LSTM model can be improved by using a attention mechanism.

7.4 Adding more training data

Although we use nearly 5000 program as our training data, it is still not enough for a deep learning issue. Therefore, it is neccessary to add more training data to avoid overfitting.

8. 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.

[11] Allamanis M, Barr E T, Bird C, et al. Suggesting accurate method and class names[C]//Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015: 38-49.