2019-5-(2): 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 [1]. 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.)

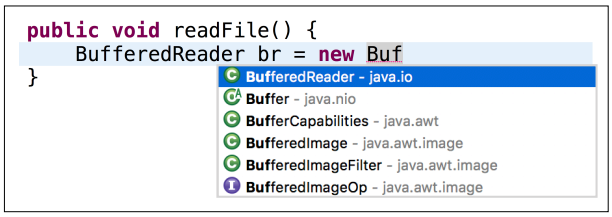


**Fig. 1** Eclipse code completion system

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 [2].

Han et al. proposed a method to improve an abbreviation-

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.

based recommendation system by using a hidden Markov model (HMM) [3]. 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 [4] and Bruch et al.'s work [5] extract a 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 [4] called GPT-2 [6]. 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 doing 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 [1] 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.

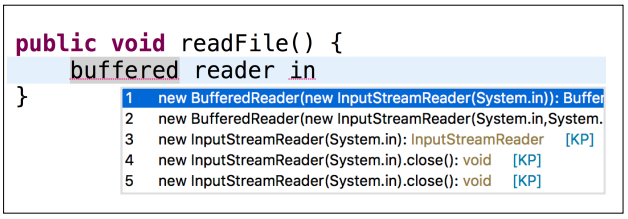


**Fig. 2** Code completion by using keyword programming

#### 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:

$$-0.05N + 1.0K - 0.01D + 0.001L \quad (1)$$

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,

[*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.

$D$ 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.

For example, the score of `array.add()` (where `array` is a local variable) with respect to the keywords `add` and `line` is calculated in this way. The above four parameters are determined as follows:

- $N = 2$
- $K = 1$ (for `add`)
- $D = 1$ (for `array`), and
- $L = 1$ (for `array`).

Thus, the score is

$$-0.05 \cdot 2 + 1.0 \cdot 1 - 0.01 \cdot 1 + 0.001 \cdot 1 = 0.891.$$

#### 2.1.5 Beam search

It is a search problem to select expressions with the highest score among all possible generations. In the previous research, they used dynamic programming to generate one result by finding the local optimal solution and $A*$ (acturally *beam search*) to get multiple generations.

Beam search is a heuristic graph search algorithm. It is based on breadth first search with width constraint. In each depth, it first sorts the candidates by a scoring function. Then remove the node with lower score and continue to search on the remaining node. *bw*, short for beam width, denotes the number of remaining nodes for each depth.

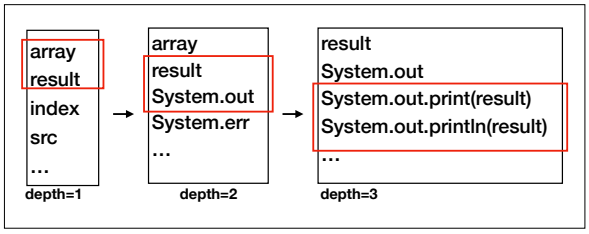


**Fig. 3** Beam search in expression generation

For example, in Fig 3, suppose the keywords are "**print out result**", and the expressions in the third black block stand for all possible expressions that the depth of AST is under 3. If the user sets the *bw* to be 2, then the system only remain two expressions before generate a deeper one. The remaining expressions are shown in the red block.

### 2.2 Neural network text generation

Our proposal in this paper uses a neural network text generation technique for improving the keyword programming. Since we use the technique by merely retargeting the domain from natural language sentences to programs, we simple overview of its functionality here.

Neural network text generation is a technique, which is originally developed in the domain of natural language processing, that can train a neural network by using a large corpus of text, so that it will generate, given an input sequence of words, a sequence of words that likely to follow the input sequence.

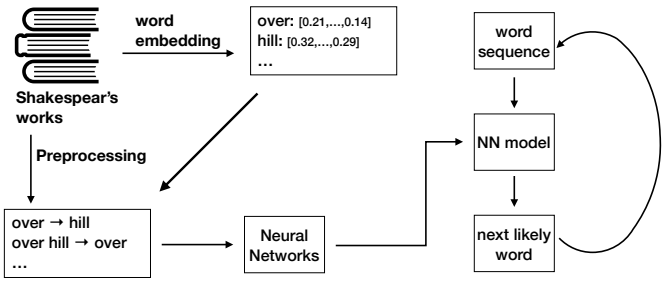


**Fig. 4** Using nerual network text generation to get a text in Shakespear's style

**Fig. 4** illustrates the three main operations in neural network text generation, namely *word embedding, training*, and *predicting*.

*Word embedding* operation converts between a word in the text and a numerical vector that is used as input and output of the generator network. Among many embedding techniques, we use an embedding method called *Word2Vec* **?**, which can naturally embed semantic similarity into the vector representation. Note that Word2Vec is based on neural network technology, we need to train another network by using a corpus. Though there are also more sophisticated and domain specific embedding methods (for example embedding nodes in a tree structure **?**), we do not consider in this paper.

*Training* a neural network is to adjust parameters in the network by providing training data set from the corpus. A training data for text generation is a sequence of words appear in the corpus as an input, and a word that appear just after the sequence as an expected output. For example, when there is a phrase "Over hill, over dale" in the corpus, there will be three training data, namely ⟨["`over`"], "`hill`"⟩, ⟨["`over`", "`hill`"], "`over`"⟩, and ⟨["`over`", "`hill`", "`over`"], "`dale`"⟩.

In our work, we use the LSTM (long short-term memory) network **?** for text generation. LSTM is one of the recurrent neural network (RNN) models. RNN is one of the network models for handling inputs in a form of a sequence by retaining a state inside of the network. LSTM is developed for handling sequences that have structures (such as natural language grammars and phrase structures) by separately maintaining the states affected by immediately preceding words and by distantly preceding words.

The trained network can *predict* the next word for a given input sequence of words. Precisely, the output of the network is probability distribution of multiple words. We can therefore determine the word with the highest probability, or a ranked list of words with higher probabilities.

The network can also be used for predicting a sequence of

words followed by the input sequence by providing the next predicted word as the next input word.

## 3. Problems

The original keyword programming does not always work well. We here point out two problematic cases, namely preference of shorter expressions and ignorance of context, and discuss the causes of the problems.

### 3.1 Preference of shorter expressions

Though it is difficult to evaluate the effectiveness of this kind of recommendation systems, the original paper **?** reported that the accuracy drops when the desired expressions get larger, even with an experiment with artificially prepared keywords. We also confirmed that the quality of recommendation degrades when we tried to input a larger expressions with our re-implementation of the original keyword programming system.

We can understand the cause of the problem from the scoring function (1) as it gives a penalty every token that does not match any keyword. It therefore ranks shorter expressions higher, if containing the same number of keywords.

We believe that this preference can be more problematic in a practical situation, for example when the programmer wants to input a long idiomatic expression. Let us see the problem by an example when we want to input the following expression that frequently appear in many programs that reads the standard input on a per-line basis.

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

Assume we provided `reader` and `in` as the keywords. Then the system ranks this expression lower with score of 1.74 than its sub-expression:

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

with score of 1.81 because the latter has a fewer number of tokens in total. By considering the fact that the latter expression alone is used less frequently, it is not ideal.

### 3.2 Ignorance of context

Even if we had improved the system to give higher scores to more frequently used expressions, another problem would remain: ignorance of context. Assume we already typed in `new BufferedReader()`, and provided `reader` and `in` as the keywords in order insert `new InputStreamReader(System.in)` as the parameter position. Since the original algorithm calculates the scores regardless the cursor position, it would then give higher score for the expression `new BufferedReader(new InputStream...)` as the parameter of `new BufferedReader( )`.

### 3.3 Summary of the problems

To summarize, we would like to improve the original keyword programming on the following two respects.

- It generally gives lower ranks to larger expressions. However, it also should rank frequently appearing expressions higher even if they are large.
- It gives the same ranking regardless the context. However, it should rank expressions higher if the expressions frequently appear in the context at the cursor position.

## 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 $+(1.74 + \textit{probability})$.

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.

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

public class Read{
    public static void main(String[] args){
        String path = "foo.in";
        reader file|
    }
}
```

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

In our proposal, we also use beam search in expression generation. However, we change the standard of the limitation.

Except for remaining the expressions with higher scores, the system also remaining the expressions with larger probability. We define $pn$ as the number of the remaining expressions compared by the probability.

To the generations in a certain depth, we first sort these by the original scoring function and remain $bw-pn$ expressions. Then sort other expressions with the new scoring function we introduced in the section 4 and remain $pn$ expressions with a higher score.

### 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. We implemented the word2vec by using the gensim package.

Subsequently, 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 [7], 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

Table 1 shows the expected expression and corresponding keyword. The first six expressions using methods or fields related to *System* class. From 7th to 10th are common expressions to process a string. The 11th and 12th are only used in a special task. The 13th and 14th are the expressions that can read a standard input from console. The last expressions has a larger abstract syntax tree.

Table 2 shows the position of the expected expression in the result and the first result by using the original keyword programming system. Table 3 shows the position and the first result by using ACKN. "×" stands for the top-30 results do not contain the expected expression.

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 6 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 key-

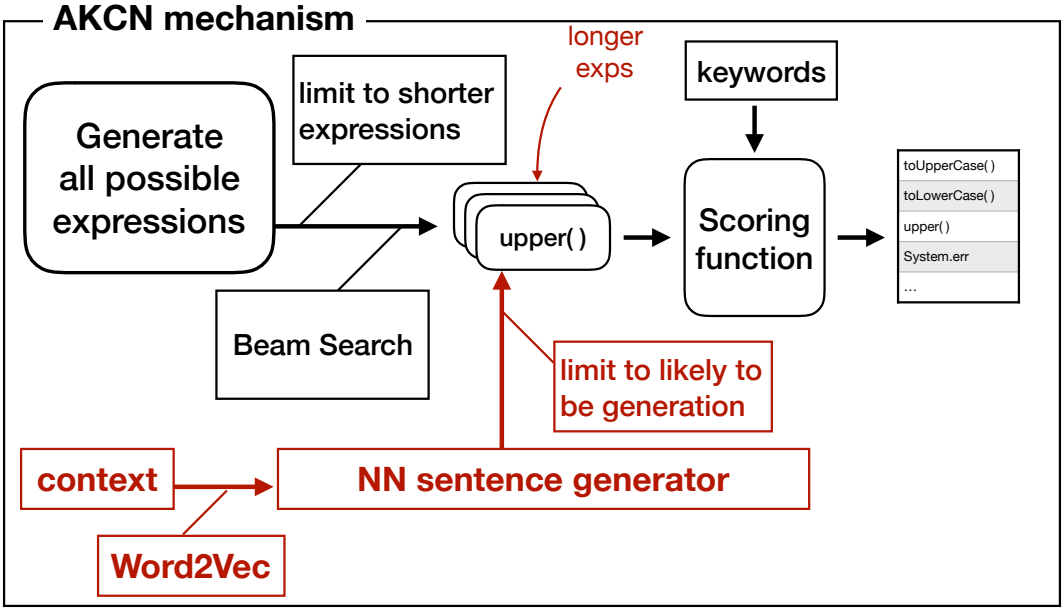


**Fig. 5** Overview of ACKN mechanism

| No. | Keyword Query | Expected Expression |
|---|---|---|
| 1 | **print result** | *System.out.println(result)* |
| 2 | **print error** | *System.err.println(error)* |
| 3 | **print result** | *System.out.print(result)* |
| 4 | **print error** | *System.err.print(error)* |
| 5 | **get time** | *System.nanotime()* |
| 6 | **get time** | *System.currentTimeMillis()* |
| 7 | **str at index** | *str.charAt(index)* |
| 8 | **upcase str** | *str.toUpperCase()* |
| 9 | **lower str** | *str.toLowerCase()* |
| 10 | **str from begin to end** | *str.substring(begin, end)* |
| 11 | **limit capacity to min** | *sb.ensureCapacity(min)* |
| 12 | **get address of host** | *InetAddress.getLocalHost()* |
| 13 | **input** | *new Scanner(System.in)* |
| 14 | **read standard in** | *new BufferedReader(new InputStreamReader(System.in))* |
| 15 | **load resource name** | *Thread.currentThread() .getContextClassLoader() .getResource(name)* |

**Table 1** keywords and expected expressions of 15 tasks

| Task | Position | Top expressions |
|---|---|---|
| 1 | × | *System.err.print(result)* |
| 2 | × | *new PrintStream(error)* |
| 3 | 2nd | *System.err.print(result)* |
| 4 | 3rd | *new PrintStream(error)* |
| 5 | 2nd | *System.getProperties()* |
| 6 | 4th | *System.getProperties()* |
| 7 | 1st | *str.charAt(index)* |
| 8 | 10th | *str* |
| 9 | 1st | *str.toLowerCase()* |
| 10 | 4th | *str.substring(end, begin).toString( )* |
| 11 | × | *sb.append(min).insert(sb.capacity(),sb.toString())* |
| 12 | 2nd | *local.getHostAddress()* |
| 13 | 24th | *new Main().readInputUntilEndOfLine()* |
| 14 | 29th | *new InputStreamReader(System.in).read()* |
| 15 | × | *ClassLoader.getSystemClassLoader() .loadClass(ClassLoader .getSystemResource(name).getRef())* |

**Table 2** position and top result by using the original keyword programming

| Task | Position | Top expressions |
|---|---|---|
| 1 | 5th | *System.err.print(result)* |
| 2 | × | *new PrintStream(error).println()* |
| 3 | 2nd | *System.err.print(result)* |
| 4 | × | *new PrintStream(error)* |
| 5 | 1st | *System.nanoTime()* |
| 6 | 1st | *System.currentTimeMillis()* |
| 7 | 1st | *str.charAt(index)* |
| 8 | 9th | *str* |
| 9 | 1st | *str.toLowerCase()* |
| 10 | × | *str.substring(end, begin).toString( )* |
| 11 | 30th | *sb.append(min).insert(sb.capacity(),sb.toString())* |
| 12 | 25th | *new Main().getLocalHost().isMulticastAddress()* |
| 13 | 22th | *new Main().readInputUntilEndOfLine()* |
| 14 | × | *new InputStreamReader(System.in,input.readLine())* |
| 15 | × | *ClassLoader.getSystemClassLoader() .loadClass(ClassLoader .getSystemResource(name).getRef())* |

**Table 3** position and top result by using the AKCN

```
public static void main(String[] args) {
    int x = 10;
    int y = 20;
    String result = (x>y)?"yes":"no";
    System.out.println(result);
    result = (x<y)?"yes":"no";
    print result
}
```

1 System.err.print(result): void [ACKN]
2 System.out.print(result): void [ACKN]
3 result: int [ACKN]
4 System.err.println(result): void [ACKN]
5 System.out.println(result): void [ACKN]

**Fig. 6** Code completion by using ACKN

word 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 sim-

ilar 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

In this paper, we propose a code recommendation system based on context-aware keyword programming. We exploit corpora by using an LSTM token generator to predict the probability of the next token and make the keyword programming recommend expressions regarding the probability. We built a Eclipse plug-in and name it ACKN to implement our idea. We evaluated our proposal by comparing the accuracy and precision of inserting 15 expressions.

### References

[1] Greg Little and Robert C Miller. Keyword programming in java. *Automated Software Engineering*, 16(1):37, 2009.

[2] Romain Robbes and Michele Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE, 2008.

[3] Sangmok Han, David R Wallace, and Robert C Miller. Code completion from abbreviated input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. IEEE, 2009.

[4] TabNine, Inc. Autocompletion with deep learning. https://tabnine.com/blog/deep/, July 15 2019. Accessed February 11, 2020.

[5] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[6] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[7] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.