Optimizing a Search-Based Code Recommendation System

Naoya Murakami
Graduate School of Arts and Sciences
University of Tokyo
Tokyo, Japan
murakami@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
Tokyo, Japan
masuhara@acm.org

***Abstract*—Search-based code recommendation systems with a large-scale code repository can provide the programmers example code snippets that teach them not only names in application programming interface of libraries and frameworks, but also practical usages consisting of multiple steps. However, it is not easy to optimize such systems because usefulness of recommended code is indirect and hard to be measured. We propose a method that mechanically evaluates usefulness for our recommendation system called Selene. By using the proposed method, we adjusted several search and user-interface parameters in Selene for better recall factor, and also learned characteristics of those parameters.**

***Keywords*-example code recommendation; integrated development environment; associative text search.**

I. Introduction

Programming to build a software system often needs to exploit libraries and frameworks with complicated application programming interfaces (APIs). In order to efficiently use those libraries and frameworks, the programmer should have different levels of knowledge from a name of a class that offers a specific functionality, to a series of method calls on several objects that accomplish a certain task.

Recommendation systems help to discover and remember knowledge about the libraries and frameworks. Many integrated development environments (IDEs) offer a code completion feature that suggests possible method and variable names or parameter types that can be filled at the cursor position.

However, such a recommendation system does not teach complicated usages consisting of multiple steps. (There are studies to teach such information [5], [6], [7], [8], [9], [10], which will be discussed in a later section.)

We believe that example programs are useful in teaching multiple-step usages, and complement existing API recommendation features. Based on this observation, we are developing Selene, a code recommendation system based on a text-based search over a large repository of open source programs [1]. Integrated with Eclipse IDE, it periodically monitors a program text being edited, and searches similar programs from a code repository. By looking relevant part in those searched programs, the programmer can find what he/she should do next.

It is however not easy to optimize Selene to recommend useful examples. This is partly because, as opposed to API recommendation, example programs give us rather indirect knowledge, whose usefulness is hard to be measured. There are also many parameters in Selene, such as the weights to tokens and the number of lines to compare, that affect search results and their visual presentation.

The paper proposes a mechanical evaluation method that estimates usefulness of Selene's recommendations by generating problems and answers from existing programs. With this method, it also optimizes the parameter set of Selene by hill-climbing.

In the rest of the paper, section II gives an overview of Selene. Section III lists the research questions that we are interested in. Section IV proposes the evaluation method that calculates recall ratios by generating problems and answers from existing programs. Section V reports the results of our experiment and some of the interesting findings. Section VI discusses related work. Finally, Section VII concludes the paper.

II. Overview of Selene

Figure 1 shows a screenshot of Selene, which works as an Eclipse IDE plug-in. The window on the left-hand side is a standard programming editor where the programmer writes a program. The three smaller windows vertically stacked on the right-hand side are the code snippets recommended by Selene.

A recommendation process in Selene basically consists of *associative search* and calculation of *local similarity scores*. It starts by sending the entire text in the editor window to a search server. The server then performs *associative search* over its code repository, and returns files that are most similar to the given text. Selene then calculates, for each line of each returned file, *a local similarity score* that denotes similarity between the text around the cursor position in the editor window and the text around the line in the returned file. It finally shows texts around the lines that have highest local similarity scores.

For the associative search, Selene uses GETA [3] with a repository of 2 million open source programs. The search algorithm is based on vector similarity of token frequencies

978-1-4673-1759-7/12/$31.00 © 2012 IEEE

RSSE 2012, Zurich, Switzerland

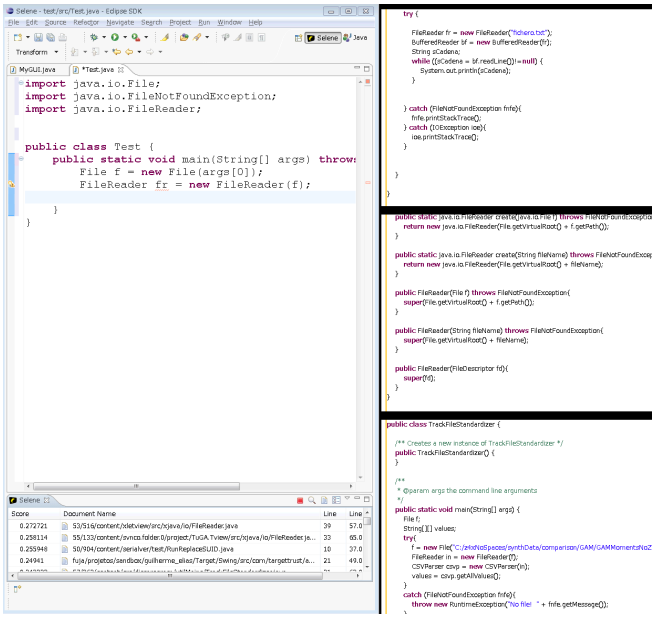

Figure 1. A Screenshot of Selene

weighted with distance from cursor position, term frequency and inverse document frequency (TF-IDF). For computing the local similarity scores, Selene implements two different algorithms, namely *the common token frequency*, which simply calculates the ratio of tokens that commonly appear in two texts, and *the pairwise sequence alignment*, which is a standard technique to compare similarity between sequences of symbols like nucleotide bases.

There are other notable features in Selene that are relevant to the paper.

- Spontaneous recommendation. Since Selene takes the entire editor text as a search query, it can start recommendation without interrupting the programmer's activity. (It also offers a shortcut key to manually start a recommendation process.)
- Efficient. Our experiment showed that a recommendation process takes 2.7 seconds on average [1].
- Text-based. Selene relies only on textual information in programs, which was crucial to achieve efficient recommendation from a large code repository. Without relying on syntactic and semantic information, Selene can support any language once we set up a code repository.

III. Research Questions

Many parameters, like weights to tokens and window sizes (number of lines to compare text fragments), in Selene were adjusted based on the authors' own experiences. However, we have many questions in order to get those parameters right.

Significance: Even if different parameters gave a different result, it is not easy to conclude that the difference is not arisen by chance, and which parameters actually contributed to the difference. We therefore need to know statistical significance of differences in results.

Locality in associative search: Selene's associative search can put more weights on the tokens close to the cursor position, or put an equal weight on all the tokens. The former configuration will retrieve files more relevant to the currently editing lines, while the latter will retrieve files that have as similar context as the one of the editing file.

Order sensitivity in contexts: Two algorithms used for scoring local similarity can be distinguished by their sensitivity of token orders. While the association search finds contexts that contain similar set of tokens disregarding the order of their occurrence, the pairwise sequence alignment can find contexts that have similar sequence of tokens. Though the latter algorithm sounds more precise, it is not clear such an algorithm can yield better results. For example, a sequence of field initialization statements for an object has no strict ordering.

Effects of comments on search: Current local similarity scoring algorithms do not distinguish comments from code. If we removed comment texts before calculating local similarity scores, does it improve search results? Since comments should also have meaningful information to the programmer, two code fragments that have more similar comments might be more useful.

Effects of comments on displaying: Current version of Selene displays code snippets as they are, including comments in the recommended code. If we removed comment texts before displaying snippets, does it improve usefulness? By removing comments, we can display more lines of code in the same area.

Balance between snippet numbers and size: How many snippets should we display on a screen? Our preliminary experiences suggest that we tend to find useful examples after browsing several recommended snippets, more snippets would improve the results. However, assuming a fixed displaying area for the entire set of snippets, more snippets shown, each snippet can show fewer lines.

IV. Mechanical Evaluation Method

In order to answer the above questions and in order to optimize Selene's parameters, we developed a method that mechanically evaluates a recall ratio as a usefulness score[1]. It first generates pairs of a query text and answer tokens from existing programs. It then lets Selene recommend code snippets with respect to a query text. It finally evaluates a recall ratio of answer tokens in the recommended snippets. Below, we explain the method on a step-by-step basis.

It first generates a *problem set* from a collection of existing program. A problem consists of a *query text* and

[1]We also measured a precision ratio, which turned out to be insignificant.

```
emailField.addKeyListener(new KeyAdapter() {                    54
  public void keyPressed(KeyEvent arg0) {                       55
    if(arg0.getKeyCode() == KeyEvent.VK_ENTER) {                56
      ActionEvent e = new ActionEvent(arg0, OK, ...             57
      informListeners(e);}}});                                  58
Dimension buttonDimension = new Dimension(73, 26);              59
JButton okButton = new JButton("OK");                           60
okButton.setPreferredSize(buttonDimension);                     61
okButton.setMnemonic('o');                                      62
okButton.addActionListener(new ActionListener() {               63
  public void actionPerformed(ActionEvent arg0) {               64
    ActionEvent e = new ActionEvent(arg0, OK, ema...            65
```

Listing 1. Example code fragment used as a query (above the horizontal bar) and an answer set (below)[3]. The boxed and underlined tokens below the horizontal bar are the answer set. The boxed ones are successfully recommended in listing 2, while the underlined ones are not.

```
if (jButtonExport == null) {                                    202
  jButtonExport = new JButton();                                203
  jButtonExport.setText("Export");                              204
  jButtonExport.setToolTipText("Exporteren");                   205
  jButtonExport.setMnemonic(KeyEvent.VK_E);                     206
  jButtonExport.setIcon(new ImageIcon(java.awt.To...            207
  jButtonExport.setPreferredSize(new java.awt.Di...             208
```

Listing 2. Example recommended snippet[5]. The tokens in boxes correspond to the boxed tokens in the answer set.

answer tokens, which are created by splitting a program text into two parts at a randomly chosen line. The upper half is a query text, which denotes a partly written program. The answer tokens are non-trivial tokens that appear in the first several lines (what we call an *answer window*) of the lower half, yet do not appear in the upper half. The answer set denotes the names that the programmer might not know, or cannot easily remember.

Listing 1 shows generation of a problem. A program text in the listing is split between lines 60 and 61. The upper half of the text (i.e., from line 1 until 60) is the query text. We could say that it represents a situation when programmer have created a button object, and were considering operations that shall be applied to the button. Assuming 5 lines for an answer window, the answer set is the tokens in boxes and with underlines. Note that other tokens such as `buttonDimension` are not in the answer set because they also appear in the upper half.

It then starts the recommendation feature in Selene with the query text, and obtains a set of recommended snippets (listing 2 is an example of such snippets for the query text in listing 1).

It calculates a recall ratio of each snippet s with respect to an answer set a by the following formula.

$$\mathrm{recall}(s, a) = \frac{|s \cap a|}{|a|}$$

[3]File LostPassFrame.java in the TijdR project (http://sourceforge.net/projects/tijdr/). The long lines are trimmed so as to fit in the paper size.

[5]File ExportPopup.java, in the OS Optiek project (http://sourceforge.net/projects/osoptiek/)

It is the ratio of answer tokens that appeared in the recommended snippet (e.g., the boxed tokens in listing 2) over the all answer tokens (e.g., the boxed or underlined tokens in listing 1). Similarly it calculates a precision ratio by $|s \cap a|/|s|$.

It calculates a recall ratio with respect to one problem by using the following formula:

$$\mathrm{recall}(S, a) = \frac{|(\bigcup_{s \in S} s) \cap a|}{|a|}$$

where a is the answer set of the problem, and S is the set of recommended snippets to the query text of the problem. Here, we assume that a token in the answer set is answered if it appears in any of the snippets.

Finally, the recall ratio with respect to a problem set is an average of a recall ratio of each problem.

V. Experiments and Findings

A. Experimental Settings

We generated 1164 problems from 60 files[6]. Each file is randomly chosen from one of 60 open source projects used in the identifier tokenization project by Butler et al. [4]. The projects cover variety of applications, including ArgoUML, Cobertura, Eclipse, FindBugs, the standard libraries in Java, JDK, Kawa and Xerces. The number of problems generated from one file is proportional to the number of lines in the file; i.e., on average, every 10 lines in a file generate one problem. When we generate a problem from a file, we randomly chose a line between 5 lines below the first class declaration and the bottom line of the file. Since lines above the first class declaration usually contain import declarations (in Java) and copyright notices, we believe that the programmer does not rely on recommendations for writing them.

When it evaluates a problem generated from a project that also included in the Selene's repository, we excluded the files in the same project from a result of the associative search. Otherwise, Selene would recommend exactly same file in the repository, which gives recall ratio close to 1.

We evaluated recall ratios by sweeping the following parameters: the algorithm for local similarity scores (either the vector similarity or the pairwise sequence alignment), whether local similarity scores are sensitive to comments, the number of recommended snippets (s), the number of lines of each snippet (ℓ) where we assumed that snippets can use 120 lines[7] on a display; i.e., $s\ell + 2(s-1) = 120$, the distance between the first line of the snippet and the most similar line, and whether comments are shown in snippets.

[6]It might give an impression that we used too small number of files, we obtained statistically meaningful results according to the t-tests.

[7]Though 120 lines of code may not easily fit in consumer-class monitors of today, we believe that many programmers will have larger, higher-resolution and multiple monitors that can accommodate those lines in near future.

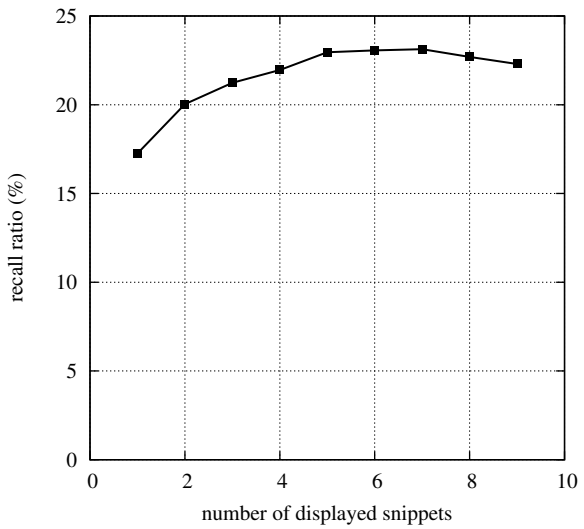

Figure 2. Recall ratios for different number of snippets. Line numbers of each snippet is adjusted so that they occupy 120 lines in total.

B. Findings

1) Statistical Significance: By evaluating recall and precision ratios for different parameters, we found that precision ratios are almost stable. This is probably because programs have to have rather obvious tokens, including control statements and primitive type names and common type names like String, which makes the ratio of useful tokens relatively insignificant. While it would need deeper analysis, we only discuss recall ratios hereafter.

For recall ratios, we carried out t-tests whenever we compared values under two parameter sets. Unless explicitly stated, we found that the differences are statistically significant with alpha level at 0.05. This suggests that the proposed evaluation method can be used for optimizing parameters in Selene.

2) Optimal Parameters: From the initial parameter set, we sought a parameter set that gives an optimal recall ratio by hill-climbing. Given a parameter set, for each parameter, we find the next value of the parameter that gives an optimal recall ratio by providing different values to the selected parameter. For example, figure 2 shows the recall ratios for different number of snippets (and different number of lines in each snippet accordingly) with a certain set of parameters. We repeated this process for three times until all the parameter values became optimal.

3) Discussion: Our experiment gave us not only an optimal set of parameters, but also several interesting observations. Due to space limitations, we here discuss a few of them.

The optimal value for the cursor distance weight suggest that the associative search should use more global information. (With larger value, it gives more weight to the tokens near the cursor position.) In other words, a program fragment written in a more similar context is more useful than a fragment that is locally similar yet written in a context less similar to the one of the currently editing file.

Table I
INITIAL AND OPTIMAL PARAMETERS.

parameter	initial	optimal
cursor distance weight (assoc.)	0.5	0.11
algorithm (local)	vector	alignment
window size (local)	10	9
weight on common tokens (local)	0.2	0.2
removal of comments (local)	no	no
# snippets $\times$ lines shown	3×38	6×18
# prior lines shown	1	1
removal of comments (display)	no	(yes)
recall ratio	17.6%	25.3%

The optimal number of snippets was 6, which is more than we expected from our past experiences. When we programed with Selene, we tended to find useful information after scrolling up/down the first few shown snippets. We therefore assumed that each snippet should be given more lines so that we can find such information without scrolling. (Note that our evaluation method only considers tokens displayed in snippets without scrolling.)

The row "# prior lines shown" suggests that we should show only *after* the lines similar to the currently editing text. However, when we gave more lines to each snippet, more lines with this parameter gave better results. Therefore, we presume that the most important fragments in the recommended code are the 18 lines below the most similar part. The similar part would be more useful than the lines beyond 18 lines.

When we evaluated the effect of comments in displayed snippets out of the hill-climbing process, we found that removal improves the recall ratio as the "removal of comments (display)" row indicates. However, we did not removed comments in the hill-climbing process as we need to use a different formula for recall calculation (namely, tokens in comments should also be removed from the answer sets).

C. Limitations

As it is mechanical evaluation, there are several limitations that we should be careful with.

- We have not verified whether recall ratios correlate with actual development times. While it would be nice to verify by carrying out a human experiment in future, we would need very careful experiment design in order to show the correlation.
- We assumed that programs were written from top to bottom, which is unlikely in many cases.
- We assumed that all tokens in the answer set are equally useful.
- We assumed that the programmer can find useful information whenever it is displayed. By considering actual programming behavior, looking into code snippets

would take a certain amount of time. We do not take such a cost into account.

VI. Related Work

There are several recommendation systems that suggest example code to the programmer, each aims at different goals and is evaluated differently. CodeBroker [5] recommends example method definitions that suites to the currently written comment text and method signature. Precision and recall evaluation is done by *manually* judging the recommendation results. Strathcona [6] recommends examples that has a similar structural context (like class hierarchy) to the currently editing one. It was evaluated through a human experiment, where subjects are requested to use Strathcona when performing designated tasks. Code Conjurer suggest examples that have a given structure and specification [7]. Since it requires the programmer to manually construct a query, it is not obvious to mechanically evaluate such a system. Since we aim at optimize parameters in a recommendation system, we rather developed a mechanical evaluation system. We believe that our approach would be useful to optimize similar systems.

Prospector [8], XSnippet [9] and RECOS [10] recommends a chain of operations to accomplish a certain task, such as creating an object of a specific class. Even though the style of recommendation is different, Selene can also suggest examples that demonstrate such a chain of operations. In fact, the set of answer tokens can be considered as an approximation of a chain of operations.

Heinemann and Hummel proposed a recommendation system that uses tokens appearing before the cursor position as contextual information of API method recommendation [11]. Though applied to recommending API methods rather than examples, their association mining algorithm has large commonality with our association search algorithm. Interestingly, they found that 2 or 3 lines around the cursor position give the best result, ours work best with 9 lines. This might be because of the sizes of the repository (i.e., 3 million *lines* of code in Heinemann and Hummel whereas 2 million *files* in ours.)

VII. Conclusion

In this paper, we proposed a method that mechanically evaluate parameters in Selene, a search-based code recommendation system. The method uses existing programs as a problem set, and calculates recall ratios in the actual recommendations. Though our work is based on Selene, we believe that the proposed method is applicable to other recommendation systems that display code fragments, such as CodeBroker and Strathcona.

Our experiments showed that the method can observe statistically significant difference in results. A hill-climbing optimization with the proposed method improved the recall ratio of Selene from 17.6% to 25.3%. We also observed several interesting characteristics of parameters, such as the optimal lines for displaying snippets.

Improving Selene's features, and making the evaluation method more reliable are two major branches of our future work. Use of syntactic and semantic information at recommendation would be one of challenges. Since we now have an evaluation method, developing such a new feature would be much easier than before. Evaluation of usefulness through human experiments is another challenge. As we presume that effects of code recommendation is usually too indirect to be observed by human experiments, we are instead planning to improve our current method by incorporating development-time information, such as the order of writing each lines in a program text.

Acknowledgement

We would like to thank Sushil Bajracharya and Cristina Lopes for their help to access the UCI Source Code Data Sets [2]. This work was partly supported as a Microsoft Research CORE Project.

References

[1] T. Watanabe and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *SUITE'11*, 2011, pp. 17–20.

[2] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI source code data sets (SDS_source-repo-18k)," 2010, http://www.ics.uci.edu/~lopes/datasets/.

[3] A. Takano, "Association computation for information access," in *Proceedings of The 6th International Conference on Discovery Science*, ser. LNCS, vol. 2843, 2003, pp. 33–44.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *ECOOP 2011*, ser. LNCS, vol. 6813, 2011, pp. 25–29.

[5] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *ICSE 2002*, 2002, pp. 513–523.

[6] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE 2005*, 2005, pp. 117–125.

[7] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, pp. 45–52, 2008.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *PLDI 2005*, 2005, pp. 48–61.

[9] N. Sahavechaphan and K. Claypool, "XSnippet: mining for sample code," in *OOPSLA 2006*, 2006, pp. 413–430.

[10] A. Alnusair, T. Zhao, and E. Bodden, "Effective API navigation and reuse," in *IEEE IRI*. 2010, pp. 7–12.

[11] L. Heinemann and B. Hummel, "Recommending api methods based on identifier contexts," in *SUITE 2011*, 2011, pp. 1–4.