SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2019, Vol. 62 052102:1–052102:16
https://doi.org/10.1007/s11432-017-9465-9

QE-integrating framework based on Github knowledge and SVM ranking

Qing HUANG[1*] & Huaiguang WU[2*]

[1]*School of Computer and Information Engineering, Jiangxi Normal University, Nanchang* 330022*, China;*
[2]*School of Computer and Communication Engineering, Zhengzhou University of Light Industry, Zhengzhou* 450000*, China*

Received 23 January 2018/Revised 9 March 2018/Accepted 11 May 2018/Published online 6 March 2019

Abstract The latest query expansion (QE) methods use the software development features for expanding queries. However, these methods allow only one feature to be considered at a time. To consider additional features simultaneously, we propose a QE method based on Github knowledge; this is a new comprehensive feature that covers both the existing features (i.e., the application program interface (API) information and crowd knowledge). It is extracted from the "pull requests" of code repositories on Github, which contain descriptions of a request and its commits, the participants' comments and the API information of the changed files. In addition, we implement a black-box framework that integrates multiple QE methods based on the support vector machine ranking called Github knowledge search repository (GKSR). Our empirical evaluation shows that the GKSR outperforms the state-of-the-art QE methods CodeHow and QECK by 25%–32% in terms of precision.

Keywords code search, query expansion, Github knowledge, SVM ranking, crowd knowledge

Citation Huang Q, Wu H G. QE-integrating framework based on Github knowledge and SVM ranking. Sci China Inf Sci, 2019, 62(5): 052102, https://doi.org/10.1007/s11432-017-9465-9

1 Introduction

The effectiveness of the code search strongly improves by using the query expansion (QE) methods [1,2]. In the last two years, many efforts have been made to build a thesaurus for software-specific words by using application program interface (API) information or crowd knowledge (CK). For example, CodeHow [3] considers the API-based feature, it expands a query with the potential APIs from online documentations. QECK [4] considers the CK-based feature, it expands a query with software-specific expansion words from the question & answer (Q&A) pairs on the stack overflow website.

Using the latest methods, however, it is possible only to consider one feature at a time. To consider more features simultaneously, we propose a QE method based on Github knowledge (GK) called Github knowledge search repository (GKSR). This method covers both the API- and the CK-based features, and develops a black-box framework based on the support vector machine (SVM) ranking[1)] that integrates multiple QE methods, such as the methods based on GK, API and CK.

The GK-based QE method has two steps: creating the GK engine and retrieving the method-level code snippets.

Creating the GK engine. GK is extracted from the "pull requests" of the code repositories on Github. First, we collected 26078 high-quality code repositories and evaluated three features ("watch",

* Corresponding author (email: qh@whu.edu.cn, hgwu@zzuli.edu.cn)
1) https://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html.

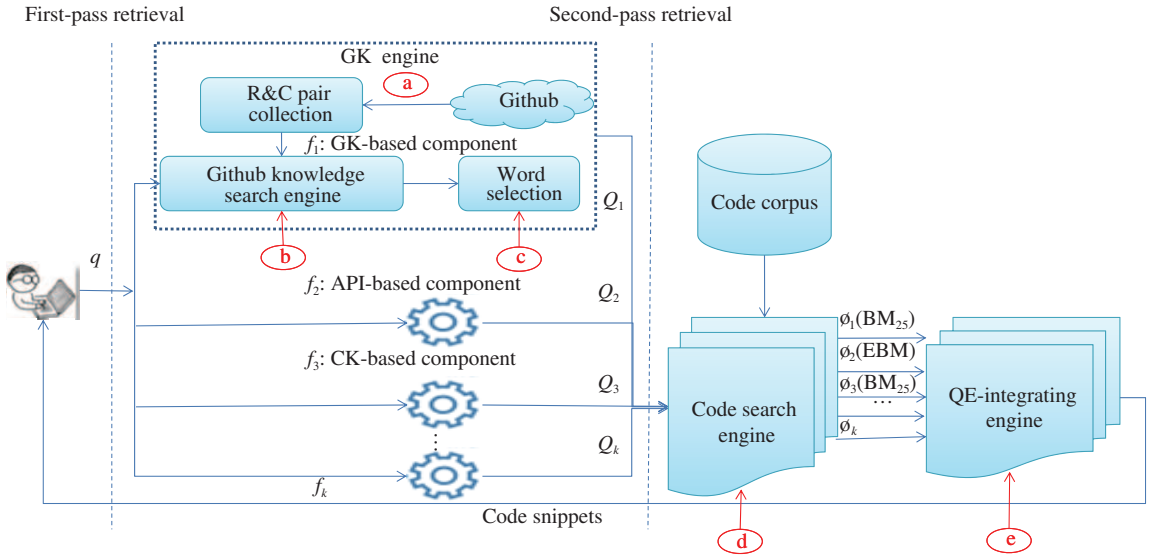

Figure 1 (Color online) Schematic of GKSR.

"star" and "fork") of each repository. For each repository, we combined each request and its corresponding conversation into a request & conversation (R&C) pair called GK. It contains CK (referring to the descriptions of the request and commits and the participants' comments) and APIs (referring to the API information of the changed files). Finally, we indexed 336014 R&C pairs with Lucene 2.9.1[2)] (see "a" in Figure 1).

Obviously. GK is more useful for software engineering tasks than either APIs or CKs used in previous studies [3, 4]. This is not only because GK covers both APIs and CK, but it is also because GK records the entire programming procedure containing programming descriptions, participants' comments about how to write or change code, and commits and changed files submitted by participants.

Retrieving method-level code snippets. To improve the possibility of retrieving relevant code snippets based on GK, we used a two-pass retrieval approach as follows.

(i) First-pass retrieval. On receiving the query q, the GK search engine calculates BM25 similarity scores [5] between a query and R&C pairs; this identifies the top m of R&C pairs (see "b" in Figure 1).

(ii) Word selection. The words selector identifies useful expansion words with a high term-frequency-inverse document frequency (TF-IDF), which generates an expanded query Q (see "c" in Figure 1).

(iii) Second-pass retrieval. The code search engine calculates similarity scores between the expanded query and code snippets in the code corpus, and it recommends the top-k code snippets (see "d" in Figure 1).

As a framework, we integrate the multiple QE methods.

Integrating QE methods with SVM ranking. To simultaneously consider more features, we integrate multiple QE methods. By considering the QE methods based on GK, API, CK and other features as k individual components $(f_1, f_2, f_3, \ldots, f_k)$, we feed a query to them and obtain k sets of query results. Then, we compute a weighted sum of k features with the SVM ranking (see "e" in Figure 1). Based on the sum, we recommend the top-k query results to users in the descending order.

We used two research questions (RQs) to evaluate the GKSR.

RQ1: Is GK effective? We compare $\text{GKSR}_{\text{noSVM}}$ (considering only the GK-based QE method and omitting the SVM ranking) against the API-based CodeHow and the CK-based QECK. Our experimental results prove that $\text{GKSR}_{\text{noSVM}}$ outperforms CodeHow and QECK by 15%–22% for precision on inspecting the top-1 query results.

RQ2: Is the QE-integrating framework effective? We compared GKSR (which involves integrating multiple QE methods and using the SVM ranking) against $\text{GKSR}_{\text{noSVM}}$. Our experimental results show

2) http://central.maven.org/maven2/org/apache/lucene/lucene-core/2.9.1/lucene-core-2.9.1.jar.

that improvement achieved by integrating the QE methods is statistically significant. The top-1 query results were 9% for precision and 16% for normalized discounted cumulative gain (NDCG).

This paper makes the following contributions.

(1) We propose a QE method based on GK. This is a new feature that covers both APIs and CK.

(2) Based on the SVM ranking, we implemented a black-box framework to integrate multiple QE methods that are based on GK, API and CK.

(3) Our experiment proves that GKSR outperforms the state-of-the-art QE methods by 25%–32% for precision.

2 Our technique

GKSR contains three engines: GK engine, code search engine, and QE-integrating engine (see Figure 1). The first two engines use the GK-based QE method. The last uses the QE methods integration.

2.1 GK engine

In this study, we have four main steps: generating, indexing, searching R&C pairs, and selecting the expansion words.

2.1.1 *Generating R&C pairs*

We scanned repositories labeled with the "C#", "Java" and "Android" tags on Github. To assure the quality of repository, only the repositories with the composite score of "watch", "star", and "fork" greater than 1 were selected. "Watch" received notifications for new pull requests and for problems related to a repository; "star" was used to star repositories for keeping track of projects; "fork" was used to make a copy of a repository. These three features reflect the attention received. People usually search for repositories that receive more attention. However, "watch", "star" and "fork" were different from one another. We normalized their scores in the range [0, 1] by using the min-max normalization method[3] [6], and we calculated the composite score as

$$\text{composite score}_i = \frac{W_i - \text{minW}}{\text{maxW} - \text{minW}} + \frac{S_i - \text{minS}}{\text{maxS} - \text{minS}} + \frac{F_i - \text{minF}}{\text{maxF} - \text{minF}}, \tag{1}$$

where W_i, S_i, and F_i represent the watch, the star and the fork scores of the i-th repository, respectively. The maximums and minimums of the watch, the star and the fork scores in all repositories, are represented by maxW and minW, maxS and minS, and maxF and minF, respectively.

To generate the R&C pairs, from each repository, we selected the "pull requests", adopted by the owner of the source code, along with the "merged pull request" tags labeled on it. Then we combined each request and its corresponding conversation into an R&C pair. From the request R, we obtained the description of the pair. From the conversation C, we obtained the participants' comments and descriptions of the relevant commits while parsing the names of the changed files as API information.

Figure 2 shows an example of an R&C pair for the pull request "move hdfs stuff out into a new contrib #34"[4]. "A" in Figure 2 shows the description of the pair; "B" in Figure 2 shows the conversation of the pair containing five participants' comments; "C" in Figure 2 shows the description of two relevant commits; "D" in Figure 2 shows the API information of 55 changed files. These four parts were combined into the text of the R&C pair, described as "move hdfs stuff contrib; hdfs client shipping issue shrink contrib require configuration less just working instead advantage core integration; move update log creation directory factory replace removed mistake; HdfsTestUtil".

3) https://github.com/AlexDem1126/Min-Max-Normalization/blob/master/minMaxNormalization.java.
4) https://github.com/apache/lucene-solr/pull/34.

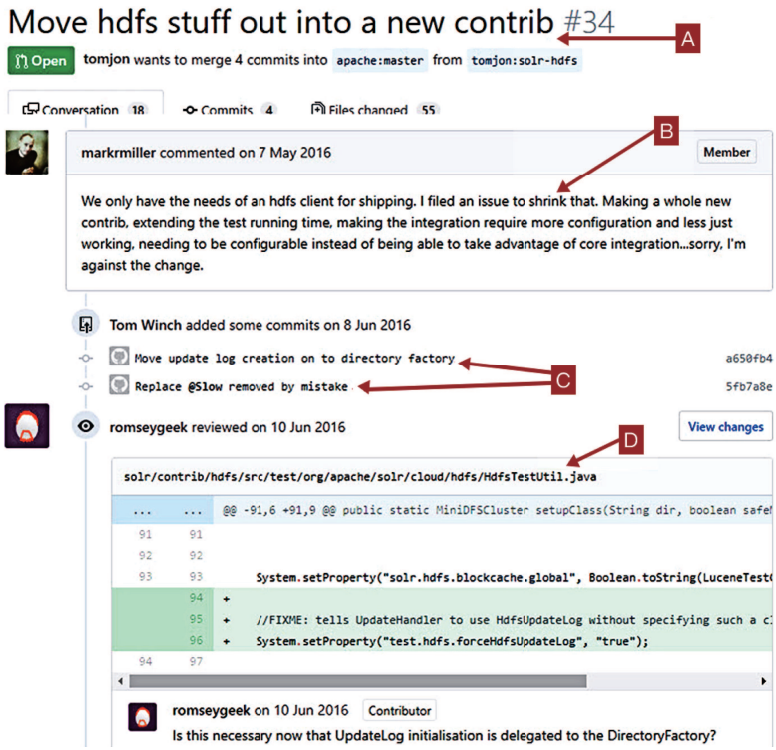

Figure 2 (Color online) Example of an R&C pair.

2.1.2 *Indexing and searching R&C pairs*

After the above steps, we achieved an R&C pair collection that contained 336014 R&C pairs. Then, we converted the text of each R&C pair into a bag of words using the text pre-processing methods [7] (e.g., standard tokenization[5)], stop-term removal[6)], identifier splitting[7)], and stemming[8)]). We indexed the pre-processed words as a document using Lucene 2.9.1 (see "a" in Figure 1).

In the first-pass retrieval of the GK-based QE method, we calculated the BM25 similarity scores between a query and the text of the R&C pairs on the index (see "b" in Figure 1). Based on the scores, we recommend the top m returned R&C pairs in descending order.

2.1.3 *Selection of word*

After obtaining relevant R&C pairs, we weighed the scores for the words used in them by using the TF-IDF weighting function [8], which identifies the high-weighting words as the expansion words. Then, we added them to the original query, and generated an expanded query (see "c" in Figure 1). We eliminated the non-discriminating words [8,9] that appeared in more than 25% of the documents in the collection.

2.2 Code search engine

2.2.1 *Creation of code corpus*

The files labeled with the ".java" and ".cs" tags were crawled from 625 open-source projects on Github. For each file, we parsed one or more method-level code snippets, and extracted the code terms with the Java development tools (JDT)[9)]. In total, we obtained 590321 code snippets. For each snippet, we

5) https://github.com/mpartel/minicompiler/blob/master/src/main/java/minicompiler/Tokenizer.java.
6) https://github.com/deeplearning4j/deeplearning4j/blob/master/deeplearning4j/deeplearning4j-nlp-parent/deeplearning4j-nlp/src/main/java/org/deeplearning4j/text/stopwords/StopWords.java.
7) https://github.com/jjfiv/galago-git/blob/master/contrib/src/main/java/org/lemurproject/galago/contrib/parse/SingleFileParser.java.
8) https://github.com/uttesh/exude/blob/master/src/main/java/com/uttesh/exude/stemming/Stemmer.java.
9) https://github.com/eclipse/eclipse.jdt.core.

Table 1 Example of integrating QE methods

Component	Query results		
GK-based component	c_1	c_2	c_3
API-based component	c_1	c_2	a_3
CK-based component	b_1	c_2	a_3

converted the code terms into a bag of words, and indexed it as a document containing two fields: the words and the snippets.

2.2.2 *Searching for code snippets*

In the second-pass retrieval of the GK-based QE method, we calculated the BM25 similarity scores between an expanded query and the words of the code snippets on the index. Based on the scores, we ranked the query results in descending order. If there was only a GK-based QE method, the top-k code snippets were recommended directly to users. If there were multiple QE methods, each method feeds its own expanded query into the code search engine that calculates the similarity scores and recommends the top-k code snippets to the QE-integrating engine (see "d" in Figure 1).

2.3 QE-integrated engine

As a black-box framework, we see the methods based on GK, API, CK and other features as k individual components without considering how they work internally. Figure 1 shows that we fed the same query q to these k components. The components based on GK, API and CK output (i) a set of query results with GK scores by using BM25 [4], (ii) a set of query results with API scores by using the extended Boolean model (EBM) [3], and (iii) a set of query results with CK scores by using BM25.

Then, we combine the k sets of query results into a single set of query results, each with k features. For each, we traverse k components. If a query result r is returned by the i-th component, its i-th feature score $f_i(q, r)$ is calculated by this component; otherwise, the score is 0.

Next, we rank the query results in the combined set with a learning-to-rank (L2R) method that computes a ranking score $f(q, r)$ for any (query q, query result r) pair, and obtains the top-n most-relevant query results (see "e" in Figure 1). In this study, we define the ranking functions as the weighted sum of k features as follows:

$$f(q, r) = w^{\mathrm{T}} f(q, r) = \sum_{i=1}^{K} w_i \times f_i(q, r), \tag{2}$$

where each feature score $f_i(q, r)$ is calculated by the i-th component. The feature weights w_i can be trained on a dataset of acquired ranking constraints by using SVM ranking, which is an L2R method.

Example. Suppose that for a query q, the components based on GK, API and CK output the code snippets: $\{R_{\mathrm{gk}} : c_1, c_2, c_3\}$, $\{R_{\mathrm{api}} : c_1, c_2, a_3\}$, and $\{R_{\mathrm{ck}} : b_1, c_2, a_3\}$, respectively.

Table 1 shows that c_1 is returned by the components based on both GK and the API, c_2 is returned by all the components, and c_3 is returned by the components based on GK. The scores of the components computed as follows:

$$\begin{aligned}
f(q, c_1) &= w_1 \times \mathrm{GKScore}(q, c_1) + w_2 \times \mathrm{APIScore}(q, c_1) + w_3 \times 0,\\
f(q, c_2) &= w_1 \times \mathrm{GKScore}(q, c_2) + w_2 \times \mathrm{APIScore}(q, c_2) + w_3 \times \mathrm{CKScore}(q, c_2),\\
f(q, c_3) &= w_1 \times \mathrm{GKScore}(q, c_3) + w_2 \times 0 + w_3 \times 0.
\end{aligned}$$

3 Evaluation

To clarify our evaluation process, we have provided a schematic representation of our operations in Figure 3. In Subsections 3.1.1–3.1.4, we describe the preliminaries of the evaluation, and propose GenQueries, GenQRPair, AutoLabel. and automatic evaluation (AE). We also introduce how to prepare the

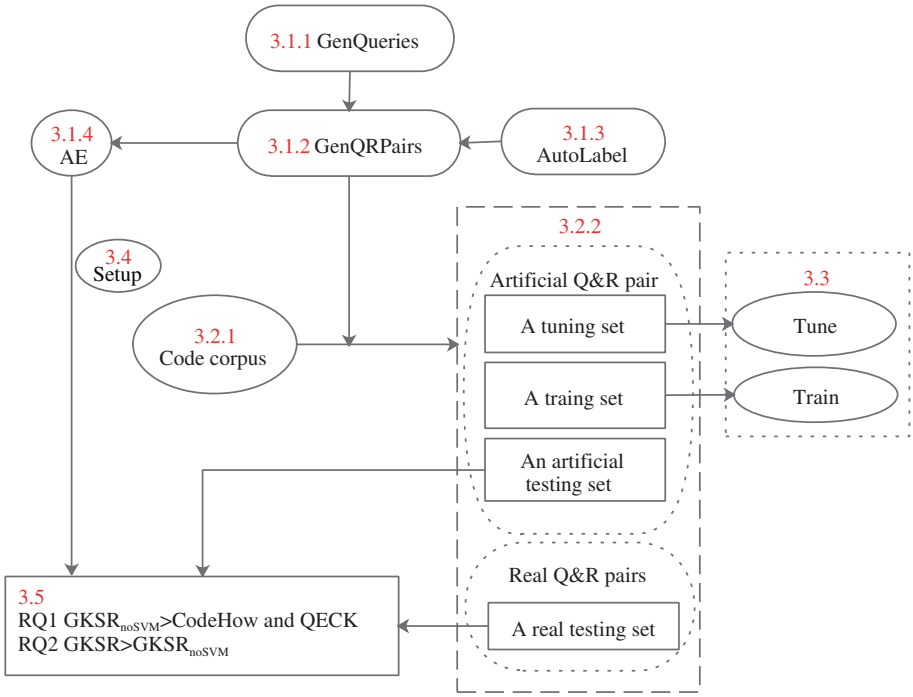

Figure 3 (Color online) Flowchart of contents of Section 3.

data in Subsection 3.2.1, especially for generating artificial Q&R pairs in Subsection 3.2.2 by invoking GenQRPair. In addition, we describe how to train and tune an L2R model based on the training and the tuning set provided in Subsection 3.3. We illustrate how to implement the tools (e.g., CodeHow and QECK) for the comparisons in Subsection 3.4. Based on the two RQs, we show the effectiveness of our method by invoking the AE proposed in Subsection 3.1.4 to test the testing set.

3.1 Preliminary

3.1.1 *Strategy for generating queries*

To obtain a large number of queries, we propose an artificial strategy for generating queries (GenQueries) that inputs a method-level code snippet m, and outputs a set of artificial queries $\{q_1, \ldots, q_k\}$. We performed the following steps. (1) We extracted the code terms from m with JDT (see "1" in "I" in Figure 4). (2) We performed a random sub-selection of the code terms (see "2" in "I" in Figure 4) by hiding the corner cases where the search engine performed particularly well or badly. Here, we selected five terms out of all the code terms as an artificial query randomly at a time. (3) We repeated the random sub-selections three times to cover the different parts of the code snippet for a complete usage. (4) Finally, we averaged the results of these artificial queries to obtain one representative prediction-quality measure.

3.1.2 *Q&R pair generation strategy*

To create a large number of benchmark datasets, we propose a generation strategy for artificial Q&R pairs (GenQRPair) This strategy inputs a query, a search mode and a ranking mode, and outputs a Q&R pair in the form of $(q, \{r_1, \ldots, r_j\})$. Here, Q refers to a query q; R refers to the ranked query results $\{r_1, \ldots, r_j\}$ with each result corresponding to a relevance rating.

(1) Perform a query in the search engine (see Figure 4) with a search mode and obtain the initial query results $\{r_1, \ldots, r_j\}$; and (2) Rank the results (see "2" in "II" in Figure 4) with a ranking mode to obtain the final results $\{r_1, \ldots, r_j\}$.

For the first step, there are many search modes, such as "Manual", "GK", "API", "CK" or "GK + API + CK". In Manual: a query is performed manually; In GK: a query is performed using the GK-based QE method; In GK+API+CK: a query is performed using the QE methods based on GK, API and CK.

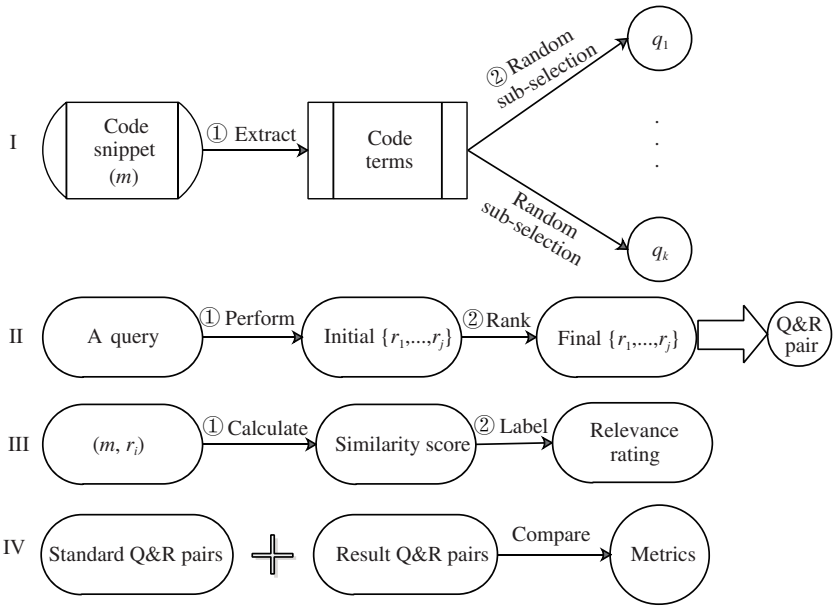

Figure 4 Evaluation strategy.

Table 2 Labeling the relevance rating

Relevance rating	Relevance score	Similarity score (%)	Tag
4	$15 = 2^4 - 1$	> 85	Most relevant
3	$7 = 2^3 - 1$	70–85	Relevant
2	$3 = 2^2 - 1$	60–70	Irrelevant
1	$0 = 2^0 - 1$	< 60	Most irrelevant

For the second step, there are many ranking modes, such as "Manual", "AutoLabel" or "ModelInfer". For Manual: the initial results are ranked manually; For AutoLabel: the initial results are ranked with the automatic labeling strategy (see Subsection 3.1.3); For ModelInfer: the initial results are ranked with the inference strategy (see Subsection 3.3).

3.1.3 *Automatic labeling strategy*

To determine whether or not two pieces of code snippets are relevant, we propose an automatic labeling strategy (AutoLabel) that inputs two pieces of code snippets m_A and m_B, and outputs a relevance rating. This process is performed in the following two steps.

(i) We calculate the similarity score between m_A and m_B as follows (see "①" in "III" in Figure 4):

$$\text{similarity}(m_A, m_B) = \frac{|\text{matchingNodes}(m_A, m_B)|}{\text{size}(m_A) + \text{size}(m_B)}, \tag{3}$$

where matchingNodes(m_A, m_B) is the number of matching AST node pairs computed by ChangeDistiller[10]. The number of AST nodes in m_A and m_B, are size(m_A), size(m_B), respectively.

(ii) Based on the similarity score, we label the relevance rating by using a four-level Likert scale (see "②" in "II" in Figure 4).

Table 2 [10] shows the guidelines for labeling the solution. For example, if the similarity score of a query result is 0.9, its relevance rating is 4, which is labeled as the "most relevant" tag.

3.1.4 *Evaluation method*

Given a Q&R pair, in the form of $(q, \{r_1, \ldots, r_j\})$, we propose an AE strategy that inputs a Q&R pair, a search mode and a ranking model, and it outputs the values of two metrics. In this strat-

10) https://bitbucket.org/sealuzh/tools-changedistiller/overview.

egy we perform following two steps: (1) We input a query in the Q&R pair, the search mode, and the ranking model into GenQRPair and obtain a Q&R' pair in the form of $(q, r'_1, \ldots, r'_j)$; and (2) we treat the Q&R pair as the "gold standard" and the Q&R' pair as the "final result", which calculates the values of the two metrics (see "IV" in Figure 4).

These two metrics are precision (P) [5] and (NDCG)[11]. P@K is the retrieval precision of the top-k query results in the ranked list, this is calculated as follows:

$$\text{P@K} = \frac{\text{\# of relevant docs in top-}k}{k}. \tag{4}$$

NDCG measures the ranking capability of the search algorithm. The algorithm is more relevant when there are more relevant results in higher positions in the hit list. NDCG is calculated as follows:

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}}, \tag{5}$$

$$\text{DCG@K} = R_1 + \sum_{i+2}^{k} \frac{R_i}{\log_2 i}, \tag{6}$$

where NDCG@K is DCG@K normalized by IDCG@K. IDCG@K is the ideal DCG@K. We sorted the results using relevance scores. R_1 is the relevance score of the 1st result. R_i is the relevance score of the i-th result.

3.2 Data preparation

3.2.1 *Code corpus*

Following the steps given in Subsection 2.2.1, we indexed 590321 method-level code snippets from the 625 open-source projects on Github. The 625 projects 6 deeplearning4j libraries[12], 8 popular libraries from [7], 295 popular Android open-source Java projects[13], 173 Google samples labeled with the "Java" tag[14], and 143 Java API examples[15].

3.2.2 *Benchmark datasets*

In this subsection, we describe how to collect artificial Q&R pairs as benchmark datasets.

Collecting artificial Q&R pairs. In the code corpus, all code snippets were sorted in a chronological order. Given the 1% most recent code snippets, we input each code snippet m into GenQueries, which generated artificial queries $\{q_1, \ldots, q_k\}$. Then we input each artificial query, the search mode of "GK + API + CK", and the ranking mode of "AutoLabel" into GenQueries, which generated a Q&R pair in the form of $(q, \{r_1, \ldots, r_j\})$. In this process, the QE methods based on GK, API and CK are considered three separate components that accept the same query q, and we calculate the GK score f_1, the API score f_2, and the CK score f_3, for each query result r_j (see Subsection 2.3). This result r_j is then labeled as AutoLabel.

To describe the above process formally, we see the query q and each query results r_j as a q&r pair, and represent it with a five-triple data (rank, qid, f_1, f_2, f_3). Here qid refers to the query q; rank refers to the relevance rating of the query result r_j; f_1, f_2 and f_3 refer to the scores that embody the features based on the GK, API and CK of the query result, respectively.

Table 3, for example, shows the query "take multiple screenshots in Android" with two corresponding query results. The first result is ranked in the second position. From f_1 (0.673), f_2 (0.725), and f_3 (0), we can see this result is returned by the two components based on GK and API. Similarly, the second result is ranked in the first position and is returned by the two components based on GK and CK.

11) https://github.com/msesmart/InformationRetrieval/blob/master/Evaluation%20of%20Language%20Models/mp2/src/edu/illinois/cs/eval/Evaluate.java.

12) https://github.com/deeplearning4j.

13) https://github.com/Trinea/android-open-project/blob/master/English%20Version/README.md.

14) https://github.com/googlesamples?language=java.

15) http://www.java2s.com/Code/JavaAPI/CatalogJavaAPI.htm.

Table 3 Data format of the q&r pair. # Query 1, take multiple screenshots in Android

rank	qid	f_1	f_2	f_3
2	1	0.673	0.725	0
1	1	0.849	0	0.784

Table 4 Benchmark dataset

Tuning set	Training set	Testing set
8850 artificial Q&R pairs	4425 artificial Q&R pairs	4425 artificial Q&R pairs; 54 real Q&R pairs

After generating 17700 artificial Q&R pairs, we split them into a tuning set (2/4 older pairs), a training set (1/4 oldest pairs), and an artificial testing set (1/4 newest pairs).

It is possible that such artificial Q&R pairs might not be convincing for some researchers. They might have the following three concerns. The first concern is that artificial queries generated from the source code would not reflect the real word world situation. To bring artificial queries close to the real queries, we the performed a random sub-selection, the number of such sub-selections, and averaged the results of the testing set to obtain one representative prediction-quality measure (see Subsection 3.1.1). Besides, a large number of artificial queries could result in valid statistical results which could reflect the real-world situations to some extent. Despite a few corner cases, the few corner cases make up only a small percentage of cases, which do not have any impact on the measured results.

The second concern is that it is not reasonable to rank the artificial query results with AutoLabel because the relevance score between a query and the query results would be calculated only with the code-term similarity between the query results and the original code snippet that generates a query (see Subsection 3.1.3). However, we consider this ranking reasonable because the underlying idea is that if a piece of code snippet cannot be found with the artificial queries generated by itself, it would also not be easily found with other queries.

The third concern is that it is not enough to consider only artificial Q&R pairs. Real evaluation supports real-world queries, but it is very time, consuming; real evaluation also limits the number of queries, narrows the types of queries and involves human subjects. Artificial evaluation could overcome these challenges, but it might suggest a higher prediction quality than what would be achieved in practice because queries from the source code tend to be overfitting. Thus, we collect real Q&R pairs to complement the artificial pairs.

Real Q&R pairs. Besides generating artificial queries from the released source code, we directly employed 54 real queries. We collected 34 real queries from [3] and 20 real queries from [4], which were ideal for CodeHow and QECK, respectively. These queries were not necessarily ideal for us. However, we still considered this a fair comparison because we need to repeat what other researchers have done.

Then we generated a real-world testing set containing 54 real Q&R pairs using GenQRPair in which the search mode and the ranking mode were "Manual". For this process, we recruited four proficient C# and Java programmers to perform real queries, and label the query results on a four-Likert scale.

In summary, Table 4 shows the benchmark dataset. We tuned the hyper-parameters of the L2R model on the tuning set, and trained the weight vector used in the ranking function on the training set. Then, we tested and reported the ranking performance on the artificial and the real testing set, respectively.

3.3 Model training and tuning

To train the L2R model, we broke each Q&R pair in the training set into small q&r pairs. With these data we learned the weight parameters of the ranking function in Eq. (2) by using the SVM ranking package [11,12]. To tune the hyper-parameters of the L2R model, we repeatedly trained using the training set and tuned on the tuning set as follows.

(1) Initially, we trained the L2R model on the training set with identical weights setting.

(2) We employed AE (see Subsection 3.1.4) that inputted each Q&R pair in the tuning set, the search mode of "GK+API+CK", and the ranking mode of "ModelInfer", and outputted the values of

two metrics. Here, the ranking mode of "ModelInfer" was an inference strategy that implied that the relevance rating was inferred based on the trained L2R model instead of the code token similarity between the original snippet and the query (as explained in Subsection 3.1.3). They are two different ranking models; therefore, the original Q&R pair is different from its corresponding Q&R' pair.

(3) We repeated the two steps given above until the values of the metrics were retained well.

3.4 Setup

3.4.1 *GKSR*

Before implementing GKSR, we had to create a collection of R&C pairs. Based on the name of each method-level code snippet in the code corpus, we indexed the 336014 R&C pairs by using Lucene 2.9.1.

GKSR had two characteristics unlike the existing QE methods: the GK-based feature and the QE-integrating framework. Thus we implemented two versions of GKSR, one without SVM ranking (referred to as $\text{GKSR}_{\text{noSVM}}$) and the other with SVM ranking.

We implemented the $\text{GKSR}_{\text{noSVM}}$ that used only the GK-based component to consider the impact of the GK-based feature on the code search. Following the steps given in Subsection 2.1.2, on receiving a query, $\text{GKSR}_{\text{noSVM}}$ retrieved the top-5 R&C pairs as the initial results from which it identified the top-9 expansion words with high TF-IDF weight. Then, it expanded an original query with these identified words, and calculated the GK scores between an expanded query and code snippets in the code corpus with BM25. Finally, it used the GK scores to directly rank the candidate code snippets. Note that because only one feature was present, we did not need to train and tune the L2R model with SVM ranking.

To consider the impact of the three features (i.e., GK, API and CK) on the code search, we implemented the GKSR that used the components based on GK, API and CK. After training and tuning the L2R model (see Subsection 3.3), GKSR calculated the GK, API and CK scores, and fed these scores to the L2R model (see Subsection 2.3). After calculating a weighted sum of the three features, it ranked the candidate code snippets based on the weighted sum scores.

3.4.2 *CodeHow*

Before re-programming CodeHow, we needed to create a collection of API descriptions. Based on the name of each method-level code snippet in the code corpus, we extracted and indexed the API names and descriptions (i.e., fully qualified name (FQN) summary and remarks) from the online documentatio[16] (i.e., "MSDN", "workbench user guide", "Java development user guide", "PDE guide", "platform plug-in developer guide" and "JDT plug-in developer guide").

CodeHow identified the top-5 relevant APIs that matched a query. We implemented CodeHow in the following steps. (1) We computed the similarity score between the API name and the query. Also, we computed the similarity score between the description and the query. (2) We combined the two scores. (3) We returned the top-5 relevant APIs. Then CodeHow added the identified APIs to an original query and generated the Boolean query expressions. Finally, CodeHow ranked the query results in the code corpus using the EBM [13].

3.4.3 *QECK*

Before re-programming QECK, we had to create a collection of Q&A pairs. Based on the name of each method-level code snippet in the code corpus, we collected the questions with the "Android, Java" tags and the accepted answer with the "AcceptedAnswer" tag from the stack exchange data dump[17]. This generated Q&A pairs which were indexed as documents consisting of words and SO scores. The words were the text of the question and the answers. The SO score is a weighted mean value between the individual scores of the question and the answers voted by the crowd.

16) http://www.eclipse.org/documentation/.
17) http://archive.org/download/stackexchange.

Table 5 Performance comparisons of three methods

Metrics	Methods	Real testing set		Artificial testing set	
		Top-1	Top-5	Top-1	Top-5
Precision	$GKSR_{noSVM}$	$\mathbf{0.757}^{+0.003,0.002}$	$\mathbf{0.743}^{+0.002,0.002}$	$\mathbf{0.771}^{+0.005,0.003}$	$\mathbf{0.756}^{+0.003,0.001}$
	QECK	0.659	0.603	0.666	0.620
	CodeHow	0.623	0.572	0.632	0.589
NDCG	$GKSR_{noSVM}$	$\mathbf{0.6914}^{+0.004,0.002}$	$\mathbf{0.6795}^{+0.003,0.002}$	$\mathbf{0.7123}^{+0.006,0.005}$	$\mathbf{0.6917}^{+0.002,0.002}$
	QECK	0.5663	0.5092	0.5785	0.5302
	CodeHow	0.5196	0.4611	0.5398	0.4821

The "+" symbol specifies results having two p-values less than 0.05. The first p-value was calculated by the pairwise comparison for $GKSR_{noSVM}$ and QECK; the second p-value was calculated by comparing $GKSR_{noSVM}$ and CodeHow.

We used the following steps to implement QECK that identified the top-5 relevant Q&A pairs that matched a query. (1) We computed an SO score and a Lucene score that showed the similarity between the Q&A pair words and the query. (2) We combined these two scores. (3) We returned the top-5 relevant Q&A pairs that matched the query. From these pairs, we extracted the top-9 software-specific words with high TF-IDF weight. (4) Finally, we expanded the original query with the identified words and ranked the query results in the code corpus with BM25.

3.5 Results and analysis

Following the steps given in Subsection 3.4, we implemented $GKSR_{noSVM}$, GKSR, CodeHow and QECK, and performed conducting experiments to answer (our two RQs). To confidently conclude that our approach was really effective, we conducted a statistical test to compare the mean values of the two metrics in the experiments. Specifically, we conducted the two-sided Wilcoxon's signed rank test between the two results. When comparing each pair of results, the primary null hypothesis was that there was no statistical difference in the performance between the two results. Here, we employed the 95% confidence level (i.e., the p-values less than 0.05 were considered significant).

RQ1: Is GK effective? Answers to this research question helped us evaluate whether or not the GK-based feature was useful for the code search. We compared the GK-based $GKSR_{noSVM}$ against the API-based CodeHow and the CK-based QECK on the real and the artificial testing set.

For the real and the artificial testing set, we employed AE (see Subsection 3.1.4) that accepted each Q&R pair in two testing sets: (i) the search mode of "GK" for $GKSR_{noSVM}$, "API" for CodeHow, "CK" for QECK, and (ii) the ranking mode of "AutoLabel" as input. Then, AE outputted the values of the two metrics.

Table 5, displays all the p-values less than 0.05. Therefore we rejected the null hypothesis, and accepted the alternative hypothesis that there was a statistically significant difference in the mean values of the two metrics. For the real testing set, the performance of $GKSR_{noSVM}$ was better than the performance of CodeHow and QECK. In terms of precision, $GKSR_{noSVM}$ was better than CodeHow by 22% P@1, and by 30% P@5.

$GKSR_{noSVM}$ was better than QECK by 15% P@1 and 23% P@5. In terms of NDCG, $GKSR_{noSVM}$ was better than CodeHow by 33% NDCG@1 and by 47% NDCG@5; $GKSR_{noSVM}$ was better than QECK by 22% NDCG@1 and by 33% NDCG@5. For the artificial testing set, the performance of $GKSR_{noSVM}$ was also better than that of CodeHow and QECK. These results are reasonable. As explained in Subsection 3.2.2 the pairs in the artificial testing set outnumbered those in the real testing set by 80 : 1. Such a large results ratio could reflect the real-world situations.

These results serve as an empirical validation of the utility of GK for the code recommendation. For example, given the query "take multiple screenshots in Android", Table 6 lists the process of matching the queries with CodeHow, QECK and $GKSR_{noSVM}$.

In Table 6, the API refers to the API descriptions from the online API documentations. CK refers to the software-specific comments. For CodeHow, only one query term "multiple" occurs 1 time on the API

Table 6 Matching queries with CodeHow, QECK, and $GKSR_{noSVM}$

	Query terms (number of times that occur)	
	APIs	Crowd knowledge
CodeHow	multiple (1)	
QECK		screenshot (76), Android (49)
$GKSR_{noSVM}$	multiple (6)	screenshot (18), Android (4)

Table 7 Performance of the baseline GKSR vs. the performance of the $GKSR_{noSVM}$

Metrics	Methods	Real testing set		Artificial testing set	
		Top-1	Top-5	Top-1	Top-5
Precision	GKSR	$\mathbf{0.822}^{+0.006}$	$\mathbf{0.804}^{+0.004}$	$\mathbf{0.832}^{+0.005}$	$\mathbf{0.803}^{+0.003}$
	$GKSR_{noSVM}$	0.757	0.743	0.771	0.756
NDCG	GKSR	$\mathbf{0.8013}^{+0.005}$	$\mathbf{0.7733}^{+0.004}$	$\mathbf{0.8189}^{+0.003}$	$\mathbf{0.7795}^{+0.003}$
	$GKSR_{noSVM}$	0.6914	0.6795	0.7123	0.6917

The "+" symbol specifies results with p-values less than 0.05 among the pairwise comparison for each QE method.

documentation "package" javax.sound.sampled[18]. For QECK, 2 query terms "Android" and "screenshot" occurred 49 and 76 times, respectively, on the Q&A pair "how to programmatically take a screenshot in Android?"[19] on stack overflow. For GKSRnoSVM, three query terms "multiple", "Android" and "screenshot" occurred 6, 18 and 4 times, respectively, on the "pull requests called run screenshot tests in multiple devices #113 of the code repositories screenshot-tests-for-android"[20] on Github. $GKSR_{noSVM}$ performed best for this query because it matched three query terms whereas CodeHow and QECK matched 1 and 2 query terms, respectively. This is reasonable because GK is a combination of API and CK. In the part "conversation" of Github requests, owners and contributors of code snippets not only discussed how to write code, but also how to submit commits to change the code.

RQ2: Is QE-integrating framework effective? Answers to this research question will shed light on whether or not the integration of multiple QE methods is useful for code search. Thus, we compared the $GKSR_{noSVM}$ (considering only the GK-based QE method and omitting SVM ranking) against the GKSR (integrating the QE methods based on GK, API and CK and using the SVM ranking) in the same retrieval scenario.

After learning the parameter vector of each ranking system (see Subsection 3.3), for the real and the artificial testing set, respectively, we employed AE (see Subsection 3.1.4) that inputted each Q&R pair in the testing set, the search mode of "GK" for $GKSR_{noSVM}$, "GK+API+CK" for GKSR, and the ranking mode of "AutoLabel" for $GKSR_{noSVM}$, "ModelInfer" for GKSR, and it outputted the values of the two metrics.

Table 7 shows all the p-values less than 0.05. It illustrates that there is a statistically significant difference in the mean values of the two metrics. For the real testing set, as compared with $GKSR_{noSVM}$, GKSR achieved 9% and 8% improvement in terms of P@1 and P@5, arespectively, and achieved 16% improvement in terms of NDCG@1. For the artificial testing set, GKSR also performed better than GKSRnoSVM. These results indicate that the integration of multiple QE methods could improve the ranking performance of a state-of-the-art approach to the code search.

Overall, Tables 5 and 7 show that GKSR outperformed CodeHow and QECK by 25%–32% in terms of precision and 42%–54% in terms of NDCG when the first query results were inspected.

18) http://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html.
19) https://stackoverflow.com/questions/2661536/how-to-programmatically-take-a-screenshot-in-android.
20) https://github.com/facebook/screenshot-tests-for-android/pull/113.

4 Discussions

4.1 Incorrect return results

Although GKSR is effective, it is still difficult to determine some code snippets. One example is the query "record audio sound". On the online API documentation called "package javax.sound.sampled"[21]. "Audio" and "sound" occurred 34 times and 4 times, respectively. However, these two words occurred only 8 times and 2 times, respectively, on the "pull requests" called "make sure recording is done in mono. #36" of the code repositories called "SoundRecorder"[22] on Github. For this query, CodeHow performed more effectively because the query terms occurred more number of times. GKSR failed because we parsed only the name of the changed files as API information instead of extracting the code terms (e.g., FQN, declaration, instantiation, and the signatures of methods invoked and filed accessed) from the method body. In our future studies, we propose to extract code terms as API information from the changed files.

4.2 Threats to validity

Threats to validity of algorithms: the GKSR, CodeHow and QECK algorithms encountered validity problems. These algorithms are unable to understand the semantic meanings of a query and the source code. For example, the results of the query "convert UTC time to local time" are actually about "convert local time to UTC time". This was because these algorithms could not distinguish the semantic meanings of different orders of words.

Threats to the validity of parameters. The parameters of CodeHow and QECK were adjusted based on [3, 4], respectively. For QECK, we set the number of the Q&R pairs and the number of expansion terms to 5 and 9, respectively. For CodeHow, we set the number of the matched APIs to 5. For GKSR, we set the number of initial R&C pairs and the number of expansion terms to 5 and 9, respectively. However, these parameters of QECK, CodeHow and GKSR were set empirically. In the future, we will experiment more with different parameters to achieve the best performance.

Threats to the impact of factors. We adjusted the training set's size, the ratio between the training and the testing sets and the hyper-parameters of SVM. Factors were routinely adjusted to the optimal state. For example, to find the optimal ratio between the training and the testing sets, we encapsulated the process of training and tuning the L2R model into a DOS function that accepted the ratio as input. Then we executed multiple such functions with the ratios 1 : 1, 1 : 2, 2 : 1, 1 : 3 and 3 : 1, by choosing the optimal ratio. However, this optimal factor always varied with various external environments. Therefore, we focused on how to adjust the factors instead of showing the so-called optimal values of the factors.

5 Related work

5.1 Free-form query search

Early code search is a coarse-grained searching that inputs a free-form query, and outputs query results (see "a" in Figure 5). Later, researchers improved it by starting from the output and the input.

After the coarse-grained searching, some researchers added the fine-grained re-ranking for the outputted query results (see "b" in Figure 5). For example, Refs. [14, 15] ranked the code examples and the code snippets, respectively.

Before the coarse-grained searching, Ref. [16] adopted pseudo-relevance feedback which reformulated the inputted query with the expansion terms from an external expansion source (see "c" in Figure 5). CodeHow [3] extracted APIs as expansion terms from online API documentations. QECK [4] extracted CK as expansion terms from Q&A on stack overflow.

GKSR is very different in the following two ways.

21) http://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/package-summary.html.
22) https://github.com/dkim0419/SoundRecorder/pull/36.

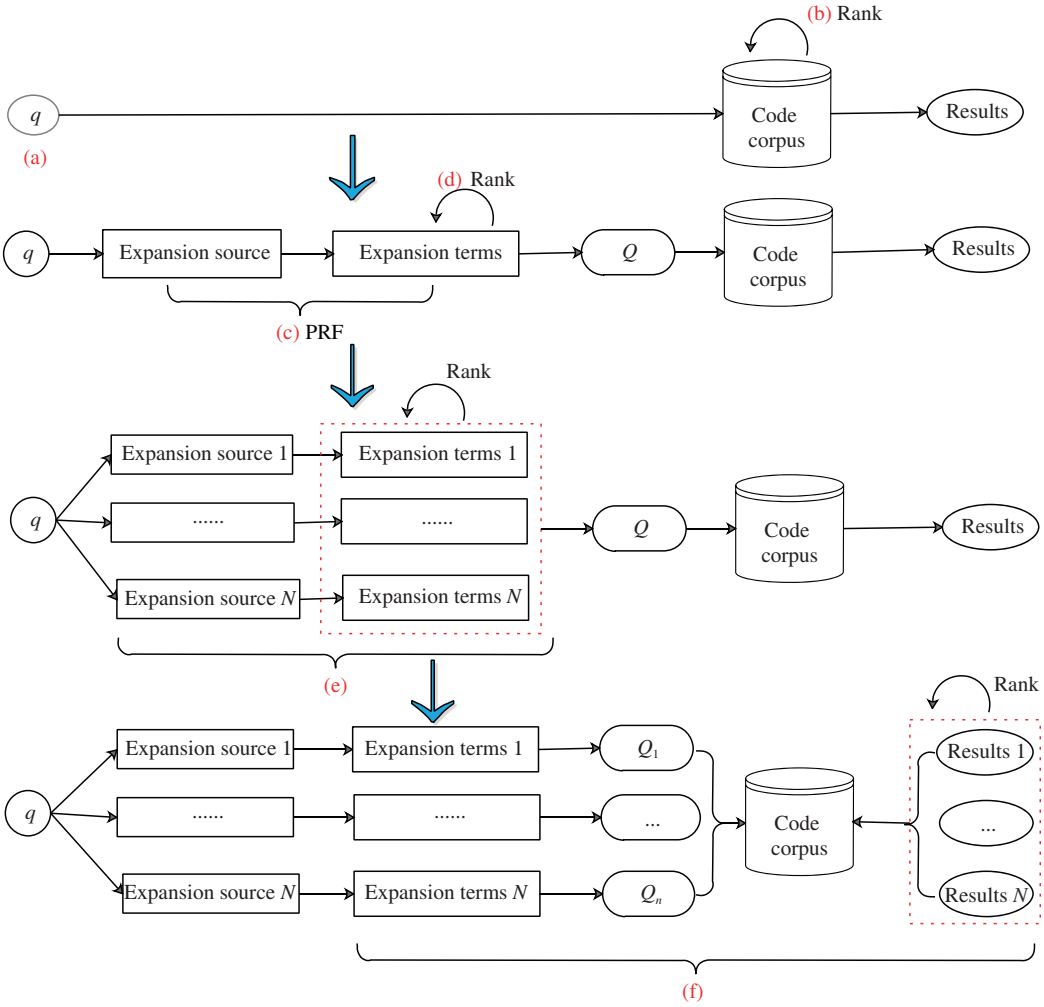

Figure 5 (Color online) Development process of the code search.

(1) The expansion source is different. CodeHow collects API information from MSDN. QECK collects CK (Q&A pairs) from stack overflow. Unlike them, GKSR collects GK from "pull requests" of code repositories on Github. GK covers both the API information and CK. In the "conversation" part of the Github request, the API information of all the changed files is the same as that of CodeHow. Also the descriptions of the request and all the commits, and the participants' comments are the same as CK of QECK.

(2) The expansion mode is different. CodeHow considers only API information and QECK considers only CK. Unlike them, GKSR integrates GK, API and CK.

To guarantee the quality of the expansion terms, Ref. [17] re-ranks and refines the expansion terms (see "d" in Figure 5). Instead of using a single external expansion source for selecting the expansion term, Ref. [16] proposed a QE method with multiple external expansion sources (see "(e)" in Figure 5). Superficially, it looks as though, we also expanded a query using the expansion terms from multiple external expansion sources, however, this is not the case. A comparison of (e) with (f) in Figure 5 shows that our method is different in the following five ways.

(1) We employed Github, stack overflow, and online documentations whereas [16] employed TREC data, Google and Derwent world patents index as the external expansion sources. (2) We did not rank multiple sets of expansion terms whereas [16, 17] did this (see a dotted box in red (e) in Figure 5). (3) We produced multiple expansion queries $[Q_1, \ldots, Q_n]$ and obtained multiple sets of query results whereas [16,17] produced an expansion query Q and obtained a single set of query results. (4) We ranked multiple sets of query results (see dotted box in red (f) Figure 5) whereas [16,17] did not rank the unique

set of query results. (5) We performed a code search whereas they performed a patent retrieval.

As shown in (f) Figure 5, GKSR is different as a black-box framework. It integrates multiple existing QE methods instead of multiple external expansion sources. By considering each QE method as an individual component, we fed a query and obtained a set of query results without considering which external expansion sources the method chose and whether it ranked the expansion terms or the query results.

5.2 Learning to rank the method

Generally, L2R methods consist of the training and the prediction. The training function inputs a training set of terms, and outputs an L2R model. The prediction function latter inputs a term and a model, and outputs the probability score of the term. All terms in the candidate set are ranked by probability scores [17].

In the training function, the training set is converted into a set of triples (q, r, V_t) [14]. Here q represents a query; r (called the term labeling) denotes the relevance between a query q and a candidate term t; and V_t (called term features) refers to a feature vector that contains the feature scores of the candidate term t. Formally, $V_t = [f_1, \ldots, f_i, \ldots, f_n]$ where f_i denotes the score of the i-th feature, and n denotes the total number of features. These triples are inputted into an L2R method (e.g., regression, RankBoost, RankSVM and LambdaMART), which obtains an L2R model. Recently such L2R methods have been applied to many free-form query search methods.

In terms of application objects, Refs. [14–17] ranked the code examples, the code snippets, expansion terms and multiple sets of expansion terms, respectively. However, in our study, we ranked multiple sets of code snippets.

In terms of term features (V_t), Ref. [14] leveraged 12 features based on the similarity, popularity, code metrics and contexts; Ref. [15] leveraged 9 features based on the text, topic and structure; Ref. [17] leverages 49 features based on the co-occurrence information of the query and an expansion term; Ref. [16] leveraged 18 features from a combination of 2 ranking methods, 3 weight evaluation methods and 3 external expansion sources. By contrast, we leveraged at least three features from the Github, CodeHow and QECK components. Note that all the existing term features referred to the statistical information about the term in a document collection. Such information lacks the skill to distinguish the semantic meanings of different orders of terms. For example, it cannot distinguish "A, B" from "B, A" based on the traditional occurrence frequencies of the terms, because the occurrence frequencies of terms ("A, B" and "B, A") are identical. In the future, we need to create a specific feature vector that represents the semantic similarity hidden deeply in a query and the source code.

In terms of the term labeling (r), the relevance of each term is labeled manually in [14–17]. Although [11] develops a method that uses clickthrough data for automatic labeling, it is difficult for most researchers to obtain valid clickthrough data without the help of major corporations. Fortunately, we have developed a method (AutoLabel) to label the relevance of each term automatically. As described in Subsection 3.2.2, the relevance score between a query and the query results was calculated only with the code-term similarity between the original code snippet m and the query results. We believe that if a piece of code snippet cannot be found with the artificial queries generated by itself, it will not be easily found with other queries either easily.

In terms of the L2R methods, Refs. [14–16] adopted RankBoost, multinomial logistic regression algorithm and LambdaMART, respectively; Ref. [17] adopted regression, RankBoost, RankSVM and LambdaMART. By contrast, our study uses SVM ranking. Note that all the existing L2R methods are machine learning algorithms. In our future research, we will adopt a deep learning algorithm to achieve a higher search quality.

6 Conclusion

To the best of our knowledge, we are the first to apply GK to QE methods. Our GKSR is the first

QE-integrating framework with SVM ranking. The experimental results verify the effectiveness of the QE methods with GKSR achieving 25%–32% improvement in terms of P@1 as compared with the state-of-the-art QE methods.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (Grant Nos. 61672470, 61640221, 61562026).

References

1 Haiduc S, Bavota G, Marcus A, et al. Automatic query reformulations fortext retrieval in software engineering. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 842–851
2 Fischer G, Henninger S, Redmiles D. Cognitive tools for locating and comprehending software objects for reuse. In: Proceedings of the 13th International Conference on Software Engineering, Austin, 1991. 318–328
3 Lv F, Zhang H Y, Lou J G, et al. CodeHow: effective code search based on API understanding and extended boolean model (E). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, 2015. 260–270
4 Nie L M, Jiang H, Ren Z L, et al. Query expansion based on crowd knowledge for code search. IEEE Trans Serv Comput, 2016, 9: 771–783
5 ManningC D, Raghavan P, Schtze H. Introduction to Information Retrieval. Cambridge: Cambridge University Press, 2008
6 de Souza L B L, Campos E, Maia M A. Ranking crowd knowledge to assist software development. In: Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, 2014. 72–82
7 Nguyen A T, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 511–522
8 Haiduc S, Bavota G, Marcus A, et al. Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 842–851
9 Gay G, Haiduc S, Marcus A, et al. On the use of relevance feedback in IR-based concept location. In: Proceedings of IEEE International Conference on Software Maintenance, Edmonton, 2009. 351–360
10 Mcmillan C, Poshyvanyk D, Grechanik M, et al. Portfolio: searching for relevant functions and their usages in millions of lines of code. ACM Trans Softw Eng Methodol, 2013, 22: 1–30
11 Joachims T. Optimizing search engines using clickthrough data. In: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, 2002. 133–142
12 Joachims T. Training linear SVMs in linear time. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, 2006. 217–226
13 Salton G, Fox E A, Wu H. Extended Boolean Information Retrieval. New York: Cornell University, 1983
14 Niu H, Keivanloo I, Zou Y. Learning to rank code examples for code search engines. Empir Softw Eng, 2017, 22: 259–291
15 Jiang H, Nie L M, Sun Z Y, et al. ROSF: leveraging information retrieval and supervised learning for recommending code snippets. IEEE Trans Serv Comput, 2017. doi: 10.1109/TSC.2016.2592909
16 Xu K, Lin H F, Lin Y, et al. Patent retrieval based on multiple information resources. In: Proceedings of the 12th Asia Information Retrieval Societies Conference, Bejing, 2016
17 Xu B, Lin H F, Lin Y. Assessment of learning to rank methods for query expansion. J Assoc Inf Sci Technol, 2016, 67: 1345–1357