Empir Software Eng
DOI 10.1007/s10664-015-9421-5

Learning to rank code examples for code search engines

Haoran Niu[1] · **Iman Keivanloo**[1] · **Ying Zou**[1]

Abstract Source code examples are used by developers to implement unfamiliar tasks by learning from existing solutions. To better support developers in finding existing solutions, code search engines are designed to locate and rank code examples relevant to user's queries. Essentially, a code search engine provides a ranking schema, which combines a set of ranking features to calculate the relevance between a query and candidate code examples. Consequently, the ranking schema places relevant code examples at the top of the result list. However, it is difficult to determine the configurations of the ranking schemas subjectively. In this paper, we propose a code example search approach that applies a machine learning technique to automatically train a ranking schema. We use the trained ranking schema to rank candidate code examples for new queries at run-time. We evaluate the ranking performance of our approach using a corpus of over 360,000 code snippets crawled from 586 open-source Android projects. The performance evaluation study shows that the learning-to-rank approach can effectively rank code examples, and outperform the existing ranking schemas by about 35.65 % and 48.42 % in terms of normalized discounted cumulative gain (NDCG) and expected reciprocal rank (ERR) measures respectively.

Keywords Learning to rank · Code example · Code search · Code recommendation

Communicated by: Denys Poshyvanyk

✉ Haoran Niu
hit.haoran@gmail.com

Iman Keivanloo
iman.keivanloo@queensu.ca

Ying Zou
ying.zou@queensu.ca

1 Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada

1 Introduction

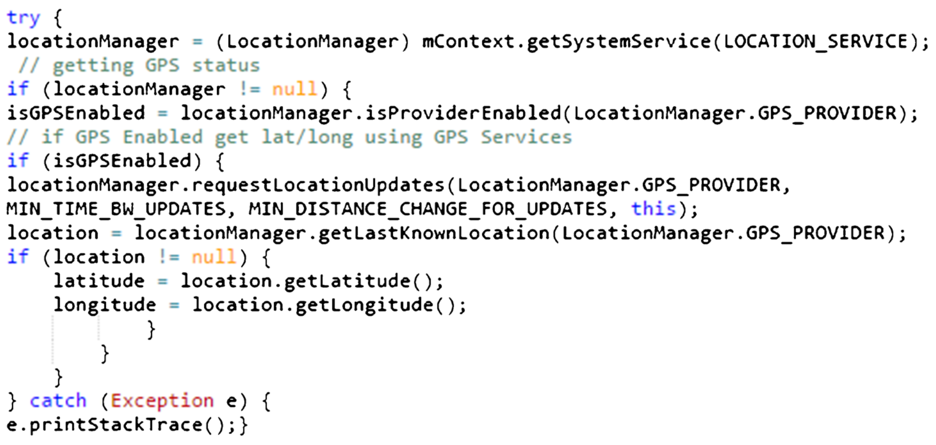
A code example is a source code snippet, i.e., a few lines of source code, used to show how a specific programming task can be implemented (Keivanloo et al. 2014). Developers rely on code examples to learn the correct way to use an unfamiliar library, framework, or Application Programing Interface (API) (Holmes et al. 2009; Robillard 2011; Stylos et al. 2009). Moreover, code examples are commonly used by developers as a form of pragmatic code reuse, which involves copying and pasting code examples from external resources (e.g., Web) into a product under development (Holmes and Walker 2012; Lange and Moher 1989; Sim et al. 2012). Code reuse also contributes to the quality of products in software development (Kapser and Godfrey 2006).

Generally speaking, developers search the Web for code examples. Such practice is referred to as Internet-scale code search (Gallardo-Valencia and Sim 2009) or simply code search (Buse and Weimer 2012). More specifically, a code search query usually consists of API tokens, i.e., class or method names. For example, if a developer wants to retrieve the last known device location from GPS using the API, $LocationManager$, he or she might use a code search engine, such as Codota[1], to search code examples that match with the query: "$LocationManager$, $getLastKnownLocation()$". Then the search process of code search engines exploits a corpus of code snippets to automatically extract the candidate code examples that match with the API tokens specified in the query. The candidate code examples are ranked based on the relevancy to the query. The studies by Brandt et al. (2009) and Sim et al. (2012) report that developers spend up to 20 % of their time searching for code examples on the Web. To provide a better support for developers to find code examples, a plethora of code search engines and recommendation systems (e.g., Kim et al. (2010); McMillan et al. (2013); Reiss (2009); Thummalapenta and Xie (2007)) are proposed to locate and rank code examples relevant to user's queries (Holmes et al. 2009).

However, not all code examples relevant to a query have equal quality (Buse and Weimer 2012; Ying and Robillard 2014). Effective code examples are expected to be concise (Buse and Weimer 2012; Kim et al. 2010), complete (Keivanloo et al. 2014), and easy to understand (Ying and Robillard 2014). Figures 1 and 2 show two candidate code examples relevant to the query "$LocationManager$, $getLastKnownLocation()$". The first candidate answer (Fig. 1) provides a complete solution as a code example. At first, the developer should create a LocationManager object to access to the system location services, and then check the status of the GPS provider. Once the GPS status is enabled, the developer can register for location updates and obtain the last known location from the GPS provider. Finally, the latitude and longitude of the location can be retrieved. However, the second candidate answer (Fig. 2) is not an effective code example since it is incomplete and contains some irrelevant code. The second candidate answer does not show how to register for location updates, and contains some unwanted code lines, such as fetching information from the network provider (highlighted in Fig. 2).

Similar to Web search, developers prefer to receive effective code examples appearing toward the top of the ranked result list (Bruch and Schfer 2008). Therefore, the ranking capability plays an important role in the success of code example search engines (Holmes et al. 2009). Several features, such as textual similarity between code examples and a query (Kim et al. 2010; McMillan et al. 2013; Thummalapenta and Xie 2007) and popularity of

[1]Codota:http://www.codota.com/



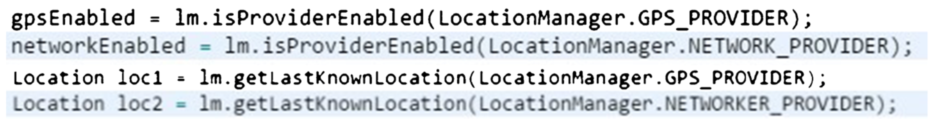
```
try {
locationManager = (LocationManager) mContext.getSystemService(LOCATION_SERVICE);
 // getting GPS status
if (locationManager != null) {
isGPSEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
// if GPS Enabled get lat/long using GPS Services
if (isGPSEnabled) {
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
MIN_TIME_BW_UPDATES, MIN_DISTANCE_CHANGE_FOR_UPDATES, this);
location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
if (location != null) {
    latitude = location.getLatitude();
    longitude = location.getLongitude();
            }
        }
    }
} catch (Exception e) {
e.printStackTrace();}
```

Fig. 1 Effective code example for the query $LocationManager, getLastKnownLocation()$

code examples (Keivanloo et al. 2014; Thummalapenta and Xie 2007) are proposed for ranking code examples. An earlier study (McMillan et al. 2013) shows that it is not sufficient to use a single feature for ranking.

A *ranking schema* specifies how to combine a set of ranking features at run-time to produce the final ranked result set (Manning et al. 2008). However, it is difficult to subjectively determine the configurations of the existing ranking schemas (i.e., the participating features or the weights of the participating features). Existing work (Panichella et al. 2013; Lohar et al. 2013) has explored to automatically calibrate the configurations using genetic algorithms. In this paper, we propose to apply learning-to-rank techniques to automatically tune the configurations of ranking schemas for code example search.

Learning-to-rank (Li 2011) is the application of machine learning algorithms in building ranking schemas for information retrieval systems. Binkley and Lawrie (2014) find that learning-to-rank can provide universal improvement in the retrieval performance by comparing learning-to-rank with three baseline configurations of a search engine.The significance of the finding is that a developer does not need to struggle with the configuration problem, but can use learning-to-rank to automatically build a ranking schema that performs better than the best configuration. Learning-to-rank has also been demonstrated to perform well in different tasks in the software engineering context, such as fault localization (Xuan and Monperrus 2014), duplication detection (Zhou and Zhang 2012), and feature traceability (Binkley and Lawrie 2014). In addition, Binkley and Lawrie (2014) show that learning-to-rank is robust. Similar to the existing ranking schemas, the learned ranking schema can be done once, off-line, as part of the model construction, and can be applied to a new dataset without a significant loss of performance.



```
gpsEnabled = lm.isProviderEnabled(LocationManager.GPS_PROVIDER);
networkEnabled = lm.isProviderEnabled(LocationManager.NETWORK_PROVIDER);
Location loc1 = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
Location loc2 = lm.getLastKnownLocation(LocationManager.NETWORKER_PROVIDER);
```

Fig. 2 Low-quality code example for the query $LocationManager, getLastKnownLocation()$

In this paper, we propose an approach that can automatically learn a ranking schema from the training data for code example search. To create the training data, we identify 12 features of 2,500 code examples for 50 queries, and collect the relevances between the queries and the corresponding code examples. Then we apply a learning-to-rank algorithm to learn how the 12 features should be combined to build a ranking schema. The learned ranking schema can be used to rank the candidate code examples for a new query at run-time.

To evaluate the performance of our learning-to-rank approach, we build a large-scale corpus of more than 360,000 code snippets crawled from 586 open-source Android projects. We address the following three research questions:

RQ1: Does the learning-to-rank approach outperform the existing ranking schemas for code example search?

We observe that the performance of our approach using the learning-to-rank technique is significantly better than the existing ranking schemas. The improvement averages 35.65 % and 48.42 % in terms of two measures of ranking quality, the normalized discounted cumulative gain (NDCG) and the expected reciprocal rank (ERR), respectively.

RQ2: Are the studied features of code examples equally important to our approach?

We have have used 8 uncorrelated features to build the ranking schema. We analyze the impact of each feature in the performance of ranking code examples. We find that similarity, frequency and context are three influential features for the ranking performance of our approach.

RQ3: Does our approach using the learning-to-rank technique outperform Codota?

Codota is the only publicly available code example search engine that is capable of ranking Android code examples. We study whether our approach can place effective code examples at the top of the ranked list by comparing our approach with Codota. The study shows that our approach can outperform Codota in recommending relevant code examples for queries related to Android application development. Specifically, for 36 out of 50 queries, our approach recommends better code example list, which is 44 % higher than Codota's winning 14 queries.

In this paper, we make the following contributions:

- Propose a code example search approach which applies machine learning techniques to automatically learn a ranking schema from training data.
- Evaluate our approach using the training and testing datasets consisting of 2,500 code examples related to 50 queries for Android application development. The evaluation results show that our approach can effectively rank candidate code examples and recommend relevant code examples for developers compared to the existing ranking schemas and online engines.

Organization of the Rest of the Paper Section 2 outlines the ranking schema for code search. Section 3 describes the details of our approach. Section 4 explains the design of the case study. Section 5 presents the results of the case study. Section 6 discusses the threats to validity. Finally, related work and conclusion are presented in Sections 7 and 8.

2 Learning-to-Rank

A ranking schema is used to estimate the relevance between a query and candidate code examples using a scoring function. As shown in (1), the scoring function is defined as a weighted sum of k (k= 12) features (Table 2), where each feature $f_i(q, c)$ measures the relevancy between a query q and a candidate code example c from the i_{th} feature's point of view:

$$rel(q, c) = \sum_{i=1}^{k} w_i \times f_i(q, c) \qquad (1)$$

Where $f_i(q, c)$ represents the i_{th} ranking feature; w_i represents the weight of the i_{th} ranking feature (Ye et al. 2014).

In this study, we attempt to train the weights automatically using the learning-to-rank technique, which is the application of machine learning algorithms in building ranking models. As stated by Li (2011), learning to rank algorithms can fall into three categories: pointwise approaches (Crammer and Singer 2001; Li et al. 2008), pairwise approaches (Burges et al. 2005; Freund et al. 2003; Herbrich et al. 2000), and listwise approaches (Cao et al. 2007; Xu and Li 2007). Pointwise approaches compute the absolute relevance score for each code example. In pairwise approaches, ranking is transformed to the classification on code example pairs to reflect the preference between two code examples. In listwise approaches, code example lists are generated through the comparison between code example pairs. In our study, it is not necessary to obtain the complete order of candidate code examples for a query, we are more interested in placing relevant code examples above less relevant ones. Thus, we apply the pairwise approach in our work. Specifically, we apply a pairwise algorithm, called RankBoost (Freund et al. 2003), to learn the ranking schema. The learning process identifies how the features should be combined in (1) to create a ranking schema for code example search.

3 Our Approach

In this section, we first introduce the overall process of our approach. Then, we present the steps for extracting features of code examples. Lastly, we show the details for learning a ranking schema from the training data.

3.1 Overall Process

The overall process of our code search approach consists of four major phases: 1) crawling Android projects; 2) extracting features; 3) learning a ranking schema; and 4) ranking candidate code examples for new queries. The first three phases are off-line processes, and are illustrated in Fig. 3. The ranking phase occurs at run-time when a query is issued by a developer. The input of our approach is the queries containing a class and one of its methods, and a corpus of code snippets. Given a query, the approach retrieves all the code snippets that contain the class or method names specified in the query from the corpus, and computes

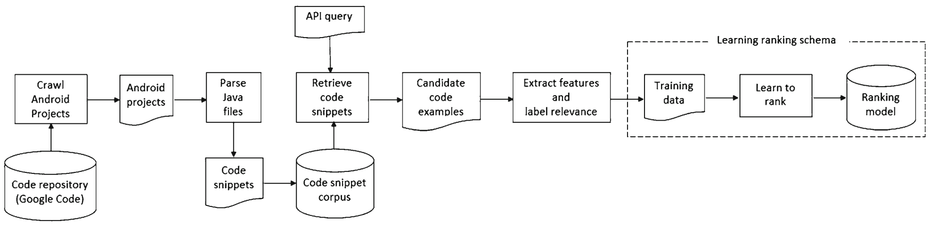


Fig. 3 The process of the proposed code example recommendation approach for building a ranking schema

Table 1 Summary of our corpus

Item	Number
Num. of projects	586
Num. of java Files	65,592
Num. of code snippets	360,068
Loc. of code snippets	3,876,295

the cosine similarities between the query and the code snippets. Finally, we select the code snippets that are the most similar to the query as the candidate code snippets. The candidates are ranked using the trained ranking schema.

Crawling Android Projects To retrieve code examples for queries, we have to prepare a corpus of code snippets. To build the corpus of code snippets for Android application development, we crawl GoogleCode[2] to find projects labeled as "Android". The crawler downloads the source code of the projects. To extract code snippets from the open-source projects, we use a Java syntax parser[3] available in Eclipse JDT to extract one code snippet from each method defined in the source code files with "java" extension. The approach of extracting code snippets has been used by the earlier work (Keivanloo et al. 2014; Mishne et al. 2012) on code example search and recommendation for Java source code. The extracted code snippets are added to the corpus. Table 1 and Fig. 4 summarize the description of the corpus used in this research.

Extracting Features We represent each candidate code example as a vector, V_s, containing a set of feature values extracted from the code example. The feature vector can be represented as $V_s = (f_1, ..., f_i, ..., f_n)$ where f_i denotes the value of the i_{th} feature, and n denotes the total number of features, i.e., 12, in our approach.

Learning a Ranking Schema In this phase, our approach automatically learns a ranking schema from the training data. The training data contains a set of queries and their candidate answers. We represent the training data as a set of triples (q, r, V_s). More specifically, q represents a query. r denotes the relevance between query q and candidate code example s. The relevance is manually labeled by curators. V_s represents a vector that contains the feature values of the candidate code example s. Then we learn a ranking schema from the training data using RankBoost (Freund et al. 2003), one of the learning-to-rank algorithm.

Ranking Candidate Code Examples for new Queries For a new query, the trained ranking schema would compute a score for each candidate code example. The score denotes the relevancy between the query and the candidate code example. All the candidate code examples are then ranked based on the computed scores in a descending order. The code examples appearing higher in the ranked result list are more relevant to the query, i.e., more likely to be effective code examples for the query.

[2]Google Code project hosting: https://code.google.com/.

[3]We use the parser from Eclipse JDT: http://www.eclipse.org/jdt/

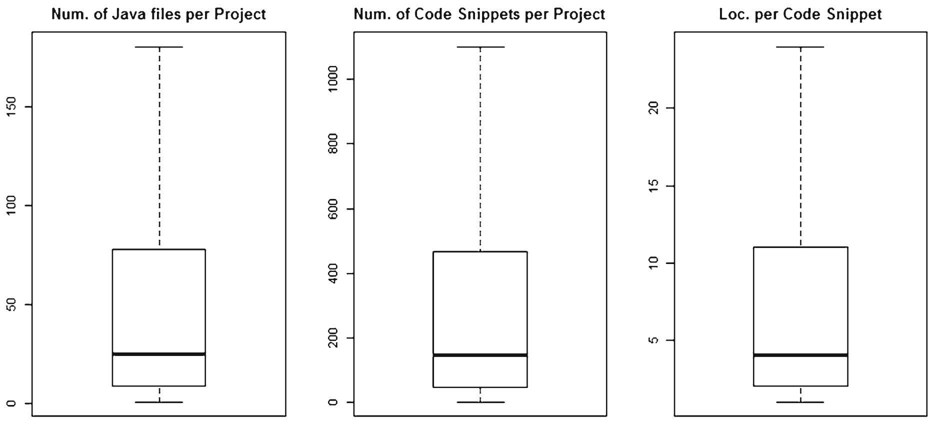


Fig. 4 Detailed summary of our corpus

3.2 Feature Extraction

In this section, we describe the features of code examples used in our approach to train the ranking schema. In total, we adopt 12 features used in earlier studies (Grechanik et al. 2012; Keivanloo et al. 2014; Thummalapenta and Xie 2007). As listed in Table 2, we classify the features in four categories: similarity, popularity, code metrics and context. Furthermore, we can divide the features into two groups: query-dependent and query-independent features (Liu 2009). Query-dependent features are calculated with regard to a query. Query-independent features only reflect characteristics of a code example regardless of the query. The 12 features are described in the following subsections.

3.2.1 Textual Similarity

Textual similarity between a query and candidate answers is the basic feature used to judge relevancy in code search (Kim et al. 2010; McMillan et al. 2013; Thummalapenta and Xie 2007). We use Vector Space Model (VSM) (Salton et al. 1975) to compute the textual similarity between a query and candidate answers. In this model, the query and the candidate code examples would be represented as vectors of term weights. Then, we compute the term weight $w_{t,d}$ for each term t using the classical term frequency-inverse document frequency ($tf-idf$) weighting schema (Manning et al. 2008). As defined in (2), $tf-idf$ weighting schema is intended to reflect the importance of a term to a document in a document collection.

$$\begin{aligned} w_{t,d} &= nf_{t,d} \times idf_t \\ nf_{t,d} &= 0.5 + \frac{0.5 \times tf_{t,d}}{max_{t\in d} tf_{t,d}} \quad idf_t = log\frac{N}{df_t} \end{aligned} \tag{2}$$

Manning et al. (2008) defines $tf-idf$ where term frequency $tf_{t,d}$ means the number of times that term t appears in document d (query or code example); document frequency df_t denotes the number of documents in which term t appears; idf_t means the inverse document frequency which represents the specificity of term t for the document that contains it. idf_t is defined as the inverse of the number of documents in which term t appears, and N is the total number of documents.

Table 2 Selected features of code examples

Group	Feature name	Feature description	Query-dependent
Similarity	Textual similarity	Cosine similarity between a query and a candidate code example	yes
Popularity	Frequency	The number of times that the frequent method call sequence of a candidate code example occurs in the corpus.	no
	Probability	The probability of following the method call sequence in a candidate code example	
Code Metrics	Line length	The number of lines of code in a candidate code example	no
	Number of identifier	The average number of identifiers per line in a candidate code example	
	Call sequence length	The number of method calls in a candidate code example	
	Comment to code ratio	The ratio of the number of comment lines to the LOC of a candidate code example (Buse and Weimer 2010)	
	Fan-in	The number of unique code snippets that call a specific candidate code example	
	Fan-out	The number of unique code snippets called by a candidate code example	
	Page-rank	The measurement of the importance of a candidate code example	
	Cyclomatic complexity	The number of decision points (for, while, etc.) in a code example	
Context	Context similarity	Jaccard similarity between the method signatures of a candidate code example and the method body where a query is issued	yes

Then we compute the textual similarity between a query and a candidate answer using cosine similarity (Salton et al. 1975) as defined in (3).

$$textualSim(V_q, V_c) = \frac{V_q^T V_c}{|V_q||V_c|} \tag{3}$$

Where V_q is a query vector and V_c is a candidate code example vector.

3.2.2 Popularity

Recent studies on code search (Thummalapenta and Xie 2007), recommendation (Buse and Weimer 2012), and completion (Ye et al. 2012) use popularity to identify candidate answers with a higher acceptance rate. Popularity represents the closeness of a code snippet to the common implementation pattern frequently observed in a corpus of source code. The underlying rationale is that the closer to the common patten in the corpus, the higher chance for the recommended code snippet to be accepted by a developer. The popularity of the underlying pattern in a code example is query-independent, and it can be evaluated using frequency (Thummalapenta and Xie 2007) or probability (Wang et al. 2013) feature.

To measure the popularity using frequency, we use the same approach suggested by Keivanloo et al.'s study (Keivanloo et al. 2014). An usage pattern is an ordered sequence of method calls that are always used together to implement a reasonable functionality. The frequency of a usage pattern is the number of times that a set of method calls in the usage pattern are used together in the corpus. The underlying usage pattern of a code example is the usage pattern most similar to the call sequences of the code example. We consider the frequency of the underlying usage pattern of a code example as the popularity of the code example. To identify the underlying usage pattern of a code example, we extract the method call sequence for each code snippet in the corpus and use the frequent itemset mining technique (Grahne and Zhu 2003) to identify the usage patterns in the corpus by analyzing the method calls that are frequently used together. Then, we identify the most similar usage pattern to the call sequence of a code example by computing the cosine similarity between them, and consider the most similar one as the underlying usage pattern of the code example.

In addition to measuring the popularity using frequency, popularity can be calculated using a probability-based approach proposed by Wang et al. (2013). We split the call sequences extracted from the corpus into pairs of two consecutive method calls. For example, a method call sequence in a code snippet can denoted as $S_m = m_1, m_2, ..., m_n$, where n represents the total number of method calls in the code snippet. Then we can split the call sequence into method call pairs $P = p_1, p_2, p_i, ..., p_{n-1}$, where p_i denotes method call pair (m_i, m_{i+1}); (m_i, m_{i+1}) means method m_i is called before m_{i+1}. After splitting all the method call sequences in the corpus into method call pairs, we can compute the probability of method call pair p_i as: $P(p_i) = \frac{1}{N}$, where N is the number of method call pairs where method m_i is called before the other method in the corpus. The probability feature of a specific code example can be computed as follows (Wang et al. 2013):

$$probability = \prod_{i=1}^{n-1} P(p_i) \tag{4}$$

Where n represents the total number of method calls in the code example, $P(p_i)$ is the probability of method call pair p_i.

3.2.3 Code Metrics

The code metric group contains four code metrics that are used in earlier studies (Buse and Weimer 2010) on code search and code quality prediction. Code metrics are a set of query-independent features. Table 2 summarizes the code metric features used in our approach. As indicated by Buse and Weimer (2010), lines of code and the average number of identifiers per line can be used to predict code readability as one of the quality aspects of code examples. Call sequence length denotes the number of method calls in the call sequence of a code example. Comment to code ratio represents the proportion of comments in the code example. Fan-in, fan-out and page-rank measures the complexity of inter-code-snippets. Fan-in is defined as the number of code snippets that call a specific code snippet. Fan-out describes the number of code snippets called by a specific code snippet. Page-rank works by counting the number links to a particular code example to determine the importance of the code snippet. The underlying assumption is that more important code snippets are likely to receive more links from other code snippets. To measure page-rank for code search

(McMillan et al. 2013), we build a graph for code snippets in the corpus based on the their call relations. In the graph, if code snippet A calls code snippet B, then, a link would exist between code snippet A and code snippet B. Then we compute the page-rank value for each code snippet in the call graph using a R package called "igraph"[4]. Assuming that code snippets, $C_1, ...C_n$, call code snippet C, and $N(E)$ represents the number of code snippets called by code snippet E. Then the package computes the page-rank of each code snippet E using (5) (Brin and Page 1998):

$$PR(E) = (1 - d) + d(PR(E_1)/N(E_1) + ... + (PR(E_n)/N(E_n)) \tag{5}$$

Where the parameter d is a damping factor, and is usually set to 0.85 (Brin and Page 1998).

3.2.4 Context Similarity

The context similarity refers to the similarity between the context of the query and the candidate code snippets (Zhong et al. 2009). An earlier study (Holmes and Walker 2005) observed that the context feature improves the success rate of code search. We use the method signatures of a code example and the method signature of the method body where a query is issued to represent the context of the code example and the query, respectively. Hence, context similarity measures the method signature similarity between candidate code examples and the method body where the query is issued (the query formulation process is described in Section 4.1.1).

We tokenize the method signatures of the method where query q is issued using camel case splitting, and represent the set containing the tokenized terms as S_q. Similarly, we denote the set containing the tokenized terms from the method signature of a candidate code example as S_c. Then, the context similarity between a query and a code examples can be computed using Jaccard index (Jaccard 1901) between the two term sets as follows:

$$contextSim(S_q, S_c) = \frac{S_q \cap S_c}{S_q \cup S_c} \tag{6}$$

Where S_q is the set containing the tokenized terms from the query context; S_c is the set containing the tokenized terms from the code example context.

For example, for the query, "LocationManager, getLastKnownLocation()", assuming that the method signature of the method body which issues the query is "private Location getLastBestLocation()", i.e., the method getLastKnownLocation() has been invoked by the class LocationManager in the method body, and the method signature of the candidate code example (shown in Fig. 1) is "public Location getLocation()", then we can tokenize the two method signatures into {private, location, get, last, best} and {public, location, get}, respectively. Therefore, the context similarity between the query and the candidate code example can be computed as 2/6 = 0.33 using (6).

3.3 Training a Ranking Schema

In our approach, we train the ranking schema using a learning to rank algorithm that is known as RankBoost proposed by Freund et al. (2003). RankBoost is an efficient boosting algorithm, so the training process on our training data just takes a few minutes. This section provides a summary of the algorithm as defined in Freund et al. (2003).

[4]Igraph package: http://cran.r-project.org/web/packages/igraph/igraph.pdf

The input of the learning-to-rank algorithm is the training data which contains the candidate code examples relevant to a set of queries. Each code example is represented as a record with the form (q, r, V_c), where q means a query id; r denotes the relevance between a query q and a candidate code example c, which is tagged by curators; and V_c is the vector containing different feature values of a code example c.

The known relevance information of code examples in the training data can be encoded as a feedback function ϕ. For any pair of code example (c_0, c_1) in the training data, $\phi(c_0, c_1)$ denotes the difference between the tagged relevance of c_0 and c_1. $\phi(c_0, c_1) > 0$ means that the code example c_1 is tagged with higher relevance than c_0. $\phi(c_0, c_1) < 0$ means the opposite. A value of zero indicates no preference between c_0 and c_1.

The learning algorithm aims to find a final ranking H that is similar to the given feedback function ϕ. To maximize such similarity, we focus on minimizing the number of pairs of the code examples which are misordered by the final ranking relative to the feedback function. To formalize the goal, let $D(c_0, c_1) = x * max\{0, \phi(c_0, c_1)\}$ so that all negative values of ϕ are set to zero. Here, x is a positive constant chosen so that $\sum_{c_0,c_1} D(c_0, c_1) = 1$. Let us define a pair (c_0, c_1) to be crucial if $\phi(c_0, c_1) > 0$. Now the goal of the learning algorithm is transformed to find a final ranking H which can minimize the (weighted) number of the crucial-pair misorderings, which is defined as *ranking loss* (Freund et al. 2003) and shown in (7).

$$loss = \sum_{c_0,c_1} D(c_0, c_1)|H(c_1) \leq H(c_0)| \tag{7}$$

Where $|H(c_1) \leq H(c_0)|$ *is 1 if* $H(c_1) \leq H(c_0)$ *holds and 0 otherwise.*

Algorithm 1 shows the details of the learning process of RankBoost (Freund et al. 2003) for the final ranking H. RankBoost operates in rounds. In each round t, it produces a ranking function f_t based on the ranking feature listed in Table 2. Meanwhile, it maintains a value $D_t(c_0, c_1)$ over each pair of code examples to emphasize the different parts of the training data. As shown in (8), the algorithm uses the ranking function f_t to update the value D_t in round t.

$$D_{t+1}(c_0, c_1) = D_t(c_0, c_1)exp(\alpha_t(f_t(c_0) - f_t(c_1))) \tag{8}$$

Where $D_t(c_0, c_1)$ *is the maintained value for each pair of code examples in round* t*;* f_t *is the produced ranking function at round* t*;* α_t *is a parameter for the ranking function* f_t*, and* $\alpha_t > 0$ (Freund et al. 2003).

Suppose we expect the code example c_0 to be ranked higher than the code example c_1, based on the definition $D_{t+1}(c_0, c_1)$ in (8), $D_{t+1}(c_0, c_1)$ will decrease if the ranking function f_t gives a correct ranking $(f_t(c_1) > f_t(c_0))$ and increase otherwise. Therefore, D_{t+1} will tend to concentrate on the misordered code example pairs. With the indication, the ranking loss of the final ranking H is proven to hold the equation (Freund et al. 2003): $loss(H) \leq \prod_{t=1}^{T} D_{t+1}$. To minimize the loss function of the ranking schema H, we have to minimize D_{t+1}. Freund et al. study (Freund et al. 2003) shows that D_{t+1} is minimized when

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1+l_t}{1-l_t}\right), \quad l_t = \sum_{c_0,c_1} D_t(c_0, c_1)(f_t(c_1) - f_t(c_0)) \tag{9}$$

Where $D_t(c_0, c_1)$ *is the maintained value for each code example pair at round* t*;* f_t *is the accessed ranking function at round* t*.*

In the process of minimizing the ranking loss of the final ranking H, the parameter α_t for the ranking function f_t is obtained using (9). Therefore, the final ranking can be represented as a weighted sum of different ranking features, i.e., $H(c) = \sum_{t=1}^{T} \alpha_t f_t(c)$, which can achieve the best performance with regard to the loss function.

Algorithm 1 RankBoost [18]

```
Require: Initial values of $D(c_0, c_1)$ over each code example pair in the training data.
1: Intialize: $D_1(c_0, c_1) = D(c_0, c_1)$
2: for $t = 1, ..., T (T = 12)$ do
3:     Build ranking function $f_t(c)$ based on the ranking feature.
4:     Choose $\alpha_t$ using Eq. 9.
5:     Update: $D_{t+1}(c_0, c_1) = D_t(c_0, c_1) exp(\alpha_t(f_t(c_0) - f_t(c_1)))$
6: end for
7: Output the final ranking $H(c) = \sum_{t=1}^{T} \alpha_t f_t(c)$
```

4 Case Study Setup

To evaluate the performance of our approach for code example search, we conduct a case study. The goal of this case study is two-fold: (1) evaluate the effectiveness of our proposed approach; and (2) compare the impact of the studied features on the performance of our approach. For the case study, we need a corpus of Android applications (i.e., source code), training and testing data, and performance measures. We have described the steps for building the corpus in Section 3.1. In this section, we discuss the methods for creating the training and testing data, the definition of the performance measures and the comparison baselines.

4.1 Training and Testing Data Collection

4.1.1 Selecting Query and Code Examples

To create the training and testing datasets, we need a set of queries and candidate code examples. We use the automatic framework proposed by Bruch and Schfer (2008) to randomly select a set of queries from our corpus. In this framework, first a code snippet is randomly selected from the corpus. The code snippet acts as an expected answer. Then, a query is automatically generated for the expected answer by randomly selecting a method call from the content of the expected answer and extracting the class and the invoked method in the method call. Our final query set used for both training and testing steps includes 50 queries along with the corresponding expected answers. The 50 queries have been listed in Table 3. The size of query set meets the acceptable number of queries required for performance evaluation of ranking schemas (Niu et al. 2012).

For a given query, the number of code snippets that contains the class or method specified in the query is very large. Building a training dataset covering all candidates requires a considerable amount of time and resources for the learning process, and it is not feasible in practice (Ye et al. 2014). Therefore, as suggested by Ye et al. study (Ye et al. 2014), as a practical solution for building the training dataset in the field of learning-to-rank, we consider only the code snippets that are the most similar to the query as the candidate code examples. This approach is also used by the earlier studies in code search engines, e.g., Bajracharya et al. (2010). For each query q in the query set, we first extract all the code snippets that contain certain items of the query, and then use the cosine similarity feature $textSim(q, c)$ to rank all the extracted code snippets. Similiar to Niu et al. study (Niu et al.

Table 3 Selected queries used for evaluation study

Id	API class	Method name
1	AnimatorSet	start()
2	WifiInfo	getIpAddress()
3	MediaPlayer	start()
4	InetAddress	getHostAddress()
5	Context	getPackageManager()
6	Socket	getInputStream()
7	SQLiteDatabase	update()
8	ContentResolver	query()
9	Camera	getParameters()
10	Message	setData()
11	MediaRecorder	start()
12	Drawable	draw()
13	ProgressDialog	show()
14	Timer	scheduleAtFixedRate()
15	HttpResponse	setStatusCode()
16	ConnectivityManager	getActiveNetworkInfo()
17	Canvas	translate()
18	Transformer	transform()
19	PowerManager	isScreenOn()
20	SensorManager	registerListener()
21	SAXParser	parse()
22	ImageView	setImageResource()
23	NotificationManager()	notify
24	Digest	update()
25	ViewPager	addView()
26	Runtime	exec()
27	SQLiteDatabase	execSQL()
28	MessageDigest	digest()
29	MenuInflater	inflate()
30	GoogleAnalyticsTracker	start()
31	Connection	prepareStatement()
32	XmlPullParser	require()
33	HttpClient	executeMethod()
34	Animation	initialize()
35	PackageManager	getPackageInfo()
36	Graphics	drawImage()
37	DatabaseHelper	getWritableDatabase()
38	URL	openStream()
39	DBAdapter	open()
40	TimePicker	setCurrentMinute()
41	Location	getLatitude()
42	UriMatcher	match()

Table 3 (continued)

Id	API class	Method name
43	Spinner	setAdapter()
44	Parcel	writeString()
45	Toast	show()
46	BluetoothAdapter	startDiscovery()
47	BluetoothServerSocket	accept()
48	HttpURLConnection	connect()
49	IBinder	transact()
50	SmsManager	divideMessage()

2012), we select the top 50 code snippets as the candidate code examples for a query. In total, we select 2,500 candidate code examples for the 50 queries.

4.1.2 Relevance Judgement

To create a training dataset for our approach to learn how to rank code examples, we need to provide the relevance between queries and their candidate code examples. The relevance between a query and a candidate answer is described by a label which represents the relevance grade. We use widely accepted multi-graded absolute relevance judgment method (Li 2011) for labeling. Specifically, we use four relevance levels, i.e., bad, fair, good, and excellent. The definition of the relevance levels is listed in Table 4. Similar to earlier studies on learning-to-rank or code search (Buse and Weimer 2012; Li 2011), we ask assessors to assign a relevance label to each pair of query and candidate answer. Assessors need to judge the relevance between each query and a candidate answer by comparing the candidate answer to the expected answer of the query. We describe the setup of relevance judgement process as follows:

Assessors To judge the relevance and get different judgements for each code example, we recruited 5 assessors to participate in our study. The 5 assessors are 3 graduates and 2 undergraduates, the 3 graduates have more than five-years' Java programming experience,

Table 4 Relevance level instruction

Relevance level	Comment
Excellent	A candidate code example is exactly matched with or highly similar to the expected answer
Good	A candidate code example contains major operations of the expected answer
Fair	A candidate code example contains a part of operations of the expected answer and contains many irrelevant lines of code
Bad	A candidate code example contains few operations of the expected answer, or is totally irrelevant to the expected answer

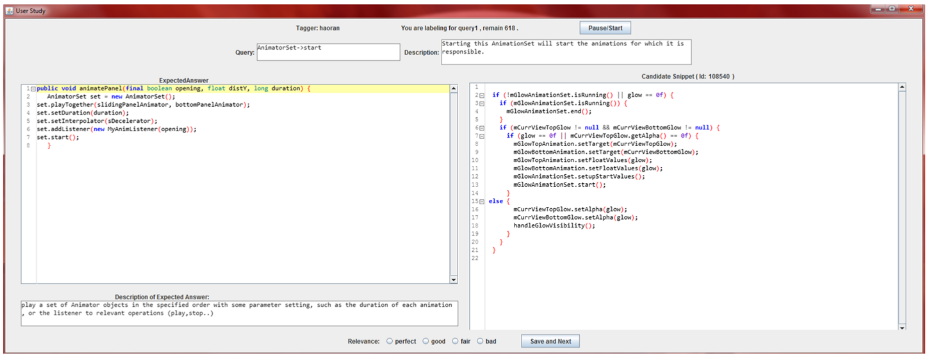
the 2 undergraduates have one-year' Android application development experience. Before the assessment process, they all received a 30-minutes' training on the usage of our labeling tool and the definitions of the four relevance levels.

Assignment To reduce the subjectivity issue in human judgement, at least three different judgements are needed to compute the agreement among assessors. As suggested by Niu et al. (2012), we divide the 50 queries into five groups and label them as G_1, G_2, G_3, G_4, and G_5. Each group contains 10 queries. Then three consecutive query groups, i.e., $\{G_1, G_2, G_3\}$, $\{G_2, G_3, G_4\}$, $\{G_3, G_4, G_5\}$, $\{G_4, G_5, G_1\}$ and $\{G_5, G_1, G_2\}$, are randomly assigned to one assessor. Therefore, the candidate answers in each group are reviewed by three assessors independently. In total, we gathered 7,500 relevance labels from the five assessors for the 2,500 candidate answers of the 50 queries in our dataset. The majority rule (Wang et al. 2013) is used when there is a disagreement on the relevance labels for a specific candidate answer. In addition, we randomly select 10 code examples out of 50 code examples (i.e., 20 %) for each query, and evaluate whether the obtained label for a code example reflects the relevancy between the code example and the query properly. We found that 92 % of the randomly selected code examples are properly labeled. It confirms the accuracy of the judgement results for our training and testing datasets.

Labeling Environment We develop a labeling tool for four-graded relevance judgment for code example search. The interface of the tool is illustrated in Fig. 5. In this tool, a query and the query description are shown on top of the window. A candidate answer (i.e., code snippet) and the expected answer of the query are shown in the main area. Four-graded buttons are displayed at the bottom of the interface. The assessors label the code examples query by query. They would not jump back and forth among different queries during the labeling process, but they could revise the final labeling results in case certain code examples are wrongly labeled. To ensure fatigue does not affect the labeling process, the labeling sessions are limited to 30 minutes to have a 10-minute break (Niu et al. 2012). On average, an assessor needs 5 sessions to label the code examples for a specific group (i.e., 10 queries).

4.2 Performance Measures

We evaluate the performance of our approach by ascertaining the goodness of the ranked results lists produced by our approach. We need the graded relevance metrics to evaluate



Tagger: haoran

You are labeling for query1 , remain 618 .

Query: AnimatorSet->start

Description: Starting this AnimationSet will start the animations for which it is responsible.

ExpectedAnswer

```
public void animatePanel(final boolean opening, float distY, long duration) {
   AnimatorSet set = new AnimatorSet();
set.playTogether(slidingPanelAnimator, bottomPanelAnimator);
set.setDuration(duration);
set.setInterpolator(sDecelerator);
set.addListener(new MyAnimListener(opening));
set.start();
   }
```

Description of Expected Answer: play a set of Animator objects in the specified order with some parameter setting, such as the duration of each animation , or the listener to relevant operations (play,stop..)

Candidate Snippet (Id: 108540)

```

 if (!mGlowAnimationSet.isRunning() || glow == 0f) {
   if (mGlowAnimationSet.isRunning()) {
     mGlowAnimationSet.end();
   }
   if (mCurrViewTopGlow != null && mCurrViewBottomGlow != null) {
     if (glow == 0f || mCurrViewTopGlow.getAlpha() == 0f) {
       mGlowTopAnimation.setTarget(mCurrViewTopGlow);
       mGlowBottomAnimation.setTarget(mCurrViewBottomGlow);
       mGlowTopAnimation.setFloatValues(glow);
       mGlowBottomAnimation.setFloatValues(glow);
       mGlowAnimationSet.setupStartValues();
       mGlowAnimationSet.start();
     }
else {
       mCurrViewTopGlow.setAlpha(glow);
       mCurrViewBottomGlow.setAlpha(glow);
       handleGlowVisibility();
     }
   }
 }

```

Relevance: ○ perfect ○ good ○ fair ○ bad

Fig. 5 Interface of relevance labeling tool

the result lists since the multi-graded relevance scale is used in the result lists. Discounted Cumulative Gain (DCG) is the only commonly used metric for graded relevance (Chapelle et al. 2009). Chapelle et al. (2009) propose another evaluation metric called Expected Reciprocal Rank (ERR) and find that ERR quantifies user satisfaction more accurately than DCG. Therefore, we use DCG and ERR to evaluate our approach. Since developers are always interested in the top k answers, we use the extended evaluation measures, k-DCG and k-ERR, to emphasize the importance of the ranking of the top k answers.

DCG (Jarvelin and Kekalainen 2002) measures whether highly relevant answers appear towards the top of ranked list. The premise of DCG is that highly relevant code examples appearing lower in the result list should be penalized. To achieve better accuracy, the DCG at each position for a chosen value of k (k-DCG) should be normalized across queries. This is done by ranking the code examples in a result list based on their relevances, producing the maximum DCG till position k, called ideal DCG (IDCG). Then, the normalized k-DCG (k-NDCG) is defined as follows. The values of k-NDCG range from 0.0 to 1.0. Higher k-NDCG values are desired.

$$k - NDCG = \frac{k - DCG}{k - IDCG}, \quad k - DCG = \sum_{j=1}^{k} \frac{2^{r_j} - 1}{\log(1 + j)} \tag{10}$$

Where r_j is the relevance label of the code example in the j_{th} position in the ranked result list; k-IDCG is the k-DCG of an ideal ordering of top k code examples, which means that top k code examples are ranked decreasingly based on their relevances (Niu et al. 2012).

k-ERR (Chapelle et al. 2009) is defined as the expectation of the reciprocal of the position k at which a developer stops when looking through a result list, which can be denoted as $\sum_{i=1}^{k} \frac{1}{k} P(user\, stops\, at\, position\, k)$. Suppose for a query, a ranked result list of k code examples, $c_1, ..., c_k$, is returned. Stopping at position k involves being satisfied with code example k, and not having been satisfied with any of the previous code examples at the positions $1, ..., k-1$, the probabilities of which can be denoted as $P(c_k)$ and $\prod_{j=1}^{k-1}(1 - P(c_j))$, respectively. The two probabilities are multiplied by $1/k$, because it is the inverse stopping rank whose expectation is computed. Therefore, the definition of k-ERR is given as (11). The values of k-ERR range from 0.0 to 1.0. Higher k-ERR values are desired.

$$k - ERR = \sum_{i=1}^{k} \frac{1}{k} P(r_i) \prod_{j=1}^{i-1} (1 - P(r_j)), \quad P(r) = \frac{2^r - 1}{2^{r_{max}}} \tag{11}$$

Where r denotes the relevance label of a code example; r_i and r_j are the relevance labels of the code examples in the i_{th} and j_{th} position respectively in the ranked result list; r_{max} is the highest relevance label in the candidate code example list. Chapelle et al. (2009)

To make sure the stability of the training process, we conducted 10-fold cross validation in the performance evaluation study. The queries are split into 10 equally sized folds $fold_1$, $fold_2$, ..., $fold_{10}$. In each round, 9 folds are the training data, and the remaining fold is the testing data. The training and testing data for each round are independent. For a given k value, we can obtain 10 k-NDCG and k-ERR values for the evaluation of a ranking schema. We compute the average of the 10 k-NDCG or k-ERR values to represent the performance of the ranking schema.

4.3 Comparison Baselines

We summarize the ranking schemas used in the existing ranking approaches into five schemas, including Random Ranking, Similarity Ranking, WeightedSum Ranking, Priority Ranking, and ReRanking. We list the five existing ranking schemas as follows:

– **Random Ranking** randomly ranks the candidate answers within a result set. It is a common practice in machine learning studies to measure the improvement over random guesses (Harrington 2012).
– **Similarity Ranking** ranks candidate code examples based on their textual similarity with the corresponding query.
– **WeightedSum Ranking** computes the relevance value between a query and a candidate code example using a weighted summation of different features of the code example with equal coefficients (Grechanik et al. 2012).
– **Priority Ranking** ranks code examples based on the primary feature. If two code examples have the same value for the primary feature, then the secondary feature is used to prioritize the two code examples (Mishne et al. 2012; Thummalapenta and Xie 2007).
– **ReRanking** would re-rank the top-*k* candidate answers determined by the primary feature using a secondary feature (Keivanloo et al. 2014).

We also compare our approach against a commercial code recommendation system, Codota. More specifically, Codota is the only publicly available code example search engine that is capable of ranking Android code examples. The ranking algorithms used in Codota identify common, credible and clear code snippets. However, the details of Codota's ranking schema is not known since it is a closed source commercial product.

5 Case Study Result

This section discusses the results of our three research questions. We describe the motivation, analysis approach and findings for each research question.

RQ1: Does the learning-to-rank approach outperform the existing ranking schemas for code example search?

Motivation A ranking schema can sort the candidate answers as a ranked list based on their relevances to the query. Existing ranking schemas are mainly hand-crafted heuristics with predefined configurations. We propose a ranking schema for code search using a learning-to-rank technique, which can automatically learn a ranking schema from a training dataset. In this research question, we evaluate whether the ranking performance of our learning-to-rank approach is better than the existing ranking schemas for code example search in the context of Android application development.

Approach At first, we conduct correlation analysis for the 12 identified features of code examples. Correlated features have similar contribution to the training process of ranking schema. Minimizing the correlation between features can reduce the time and resource of training process and increase the stability of the trained ranking schema (Shihab et al. 2010). Spearman's rank correlation coefficient (Spearman's rho) and Pearson product-moment correlation coefficient (Pearson's r) are two commonly used measures for the correlation analysis. Pearson's r evaluates whether two variables tend to change together at a constant rate. When the relationship between the variables is not linear, Spearman's rho is more

Table 5 Mapping Spearman's rho with coefficient level (Campbell and Swinscow 2009)

Spearman's rho	Coefficient level
over 0.8	very high
0.6 - 0.8	high
0.4 - 0.6	normal
0.2 - 0.4	low
less than 0.2	very low

appropriate to use. Since the different feature values of code examples are not always change with a consistent rate, we use Spearman's rho to analyze the correlation between different features of code examples. The values of Spearman's rho range from -1 and +1. A value close to +1 or -1 means that one variable is a monotone function of the other. A value close to 0 indicates that the two variables have no correlation. Table 5 shows the mapping between the values of Spearman's rho and the level of correlation (Campbell and Swinscow 2009). After finding the highly correlated features, we select one representative feature from the correlated features.

Once we narrow down to minimally collinear features, we use these features to train a ranking schema using our approach, and then compare the ranking performance of our approach with the existing ranking schemas as described in Section 4.3. We use the corpus and dataset described in Section 4.1 for this study. To evaluate our approach, we conducted 10-fold cross validation (described in Section 4.2) to obtain k-NDCG and k-ERR values as defined in (10) and (11) respectively. Higher k-NDCG and k-ERR values are desired. We compute the performance improvement of our approach by comparing the performance of our approach with baseline ranking schemas using the formula: $Improvement = \frac{O-B}{O}$, where O denotes the ranking performance of our approach, B means the ranking performance of a baseline ranking schema. To ensure the reliability of our results, we did sensitivity study, with k ranging from 1 to 20.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data (Sheskin 2007). We use Mann-Whitney U test to determine if the observed difference in the performance of our approach and the existing ranking schemas is significant or not. We conduct Mann-Whitney U test with 5% confidence level (i.e., p-value $<$ 0.05). We also calculate effect size using Cliff's delta (Romano et al. 2006) to quantify the difference between our approach and the existing schemas. Cliff's delta is -1 or +1 if all the values in one distribution are larger than the other one, and it is 0 when two distributions are identical (Cliff 1993). Cohen's standards are commonly used to interpret effect size. Therefore, we map the Cliff's delta to Cohen's standards as summarized in Table 6.

To test whether a limited training dataset can significantly affect the ranking performance of the learned ranking schema, we used a subset of the entire dataset (i.e., totaling 50 queries) to train a ranking schema. The subset contains 10 queries and their candidate code

Table 6 Mapping Cliff's delta with Cohen's standards (Romano et al. 2006)

Cliff's delta	Cohen's d	Cohen's standards
0.147	0.2	small
0.330	0.5	medium
0.474	0.8	large

examples (i.e., 500 code examples). To avoid the bias incurred by the fact that training and testing data are in the same dataset used for the 10-fold cross validation, we created a new testing dataset which is not included in the 50 queries used for the 10-fold cross validation. The new testing dataset contains 5 new queries and their candidate code examples (i.e., 250 code examples) extracted from our corpus as described in Section 4.1.1. The ranking of the code examples for each new query is generated using the ranking schema trained using the 10 queries (i.e., the subset of the 50 queries). To examine the correctness of the generated ranking, we asked one assessor to label the relevance between the 5 new queries and their candidate code examples. We compare the generated ranking and labeled ranking by computing k-NDCG and k-ERR values to evaluate the performance of the ranking schema that is trained using 10 queries and tested using 5 new queries.

Result Figure 6 shows the correlation structure of the 12 identified features. As indicated by Table 5, five features, including line length, call sequence length, fan-in, fan-out and cyclomatic complexity are correlated. Since line length is commonly used to judge about code quality, e.g., Buse and Weimer (2010); Nagappan and Ball (2005), we select line length to represent the five correlated ones. Therefore, we have 8 code example features to build the ranking schemas, that is, the features in Table 2 except for call sequence length, fan-in, fan-out and cyclomatic complexity.

Figures 7 and 8 present the k-NDCG and k-ERR results for our approach and five baselines, with k ranging from 1 to 20. We can observe from Figs. 7 and 8 that our approach

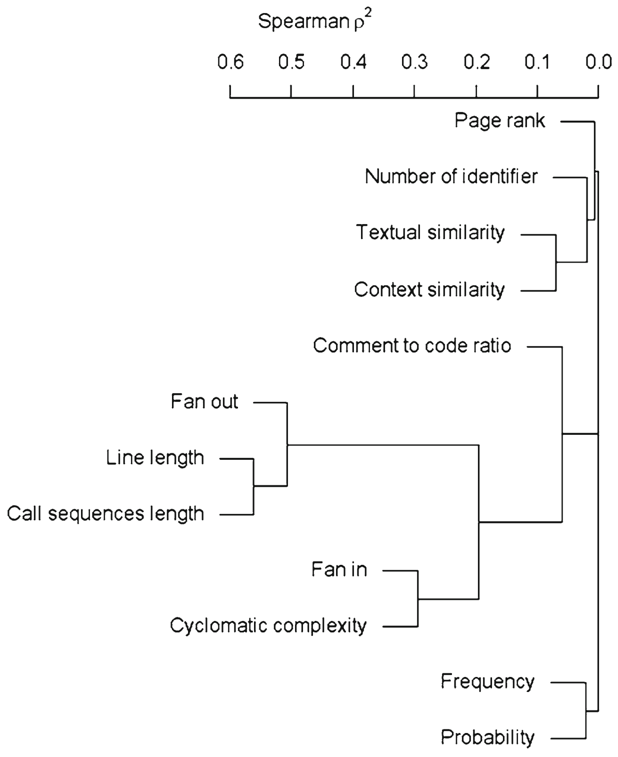


Fig. 6 The correlation structure of the 12 identified features

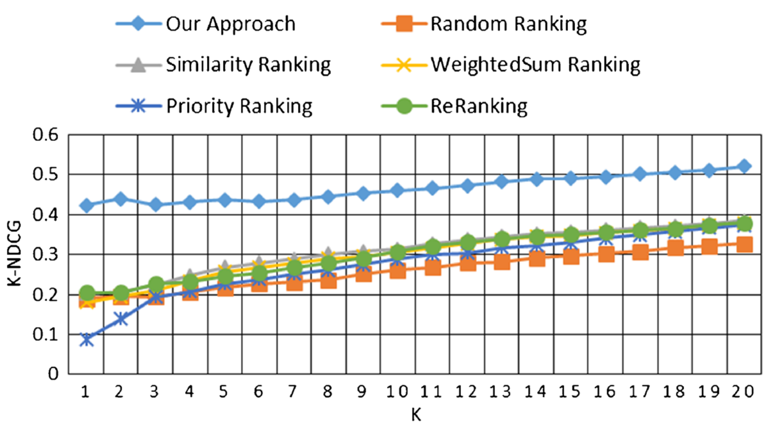


Fig. 7 The *k*-NDCG comparison result between our approach and the existing ranking schemas

achieves better performance than the other ranking schemas on both *k*-NDCG and *k*-ERR. The *k*-NDCG improvement achieved by our approach ranges from 32.37 % to 54.61 %. The *k*-ERR improvement achieved by our approach ranges from 43.95 % to 51.66 %. Considering the fact that most search engines always show top 10 results in their first result page (McMillan et al. 2013), we care more about the top 10 results (i.e., $k = 10$). As shown in Figs. 9 and 10, 10-NDCG and 10-ERR results for our approach are 0.46 and 0.38, which are better than all of the studied baselines, i.e., Random Ranking (0.26 and 0.22), Similarity Ranking (0.31 and 0.19), WeightedSum Ranking (0.31 and 0.18), Priority Ranking (0.29 and 0.19), ReRanking (0.31 and 0.2). Tables 7 and 8 list the performance improvement of our approach, *p*-values and effect size of performance comparison between our approach and baselines using *k*-NDCG and *k*-ERR measures with k=10. On average, our approach achieved 35.65 % and 48.42 % improvement than the existing ranking schemas for 10-NDCG and 10-ERR respectively. Applying Wilcox rank sum test and Cliff's delta shows that the improvement achieved by our approach is significant with large effect size. The 10-NDCG and 10-ERR values obtained for the ranking schema trained from a subset

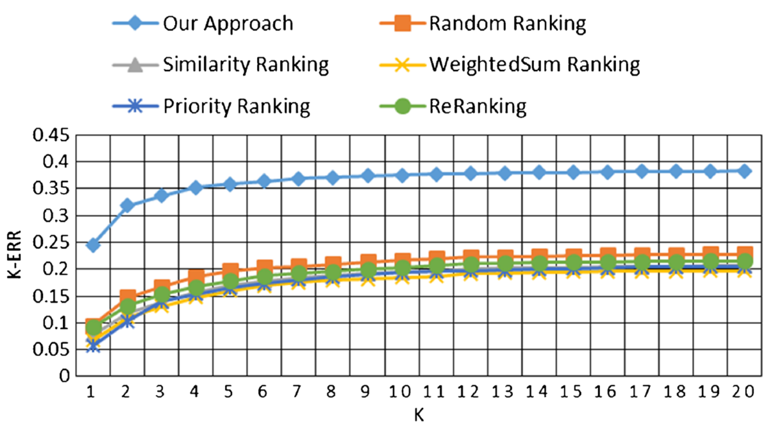


Fig. 8 The *k*-ERR comparison result between our approach and the existing ranking schemas

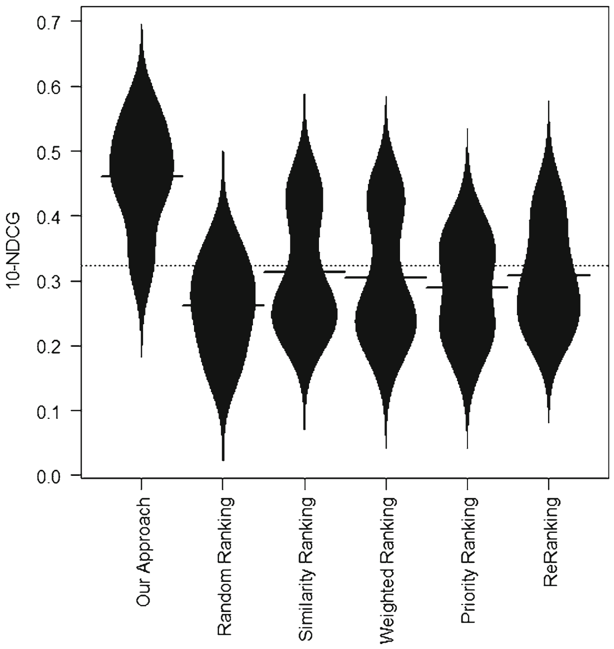


Fig. 9 The result of performance evaluation study between our approach and the existing ranking schemas in terms of 10-NDCG

(10 queries) and tested on 5 new queries are 0.45 and 0.36, respectively, which are similar to the ranking performance obtained using the entire training dataset (45 queries) and 10-fold cross validation. Even with the smaller training dataset, we can observe that the learning-to-rank approach does perform well in ranking code examples for new queries.

Our learning-to-rank approach can outperform the existing ranking schemas in ranking code examples, with an average improvement of 35.65% and 48.42% in terms of 10-NDCG and 10-ERR respectively.

RQ2: Are the studied features of code examples equally important to our approach?

Motivation Our approach uses 8 uncorrelated features used in earlier studies to learn how to rank candidate answers for code example search. We include the features since earlier studies reported an improvement on the ranking capability when the features are combined with similarity feature (e.g., Bajracharya et al. (2006); Grechanik et al. (2012); Thummalapenta and Xie (2007)). In this research question, we evaluate the impact of each feature in the performance of ranking code examples using our approach.

Approach We consider the ranking schema built using our approach with 8 uncorrelated features, as the baseline ranking schema s^{base}. Then, we build alternative ranking schemas to analyze the impact of each feature on the ranking performance. We use the same approach as Breiman study (Breiman 2001) to build alternative ranking schemas. We randomly order the values of one feature for the candidate code examples each time when building ranking schemas using our approach. The rationale is that the impact of one feature on the performance of the baseline approach can be observed when the feature values are randomized.

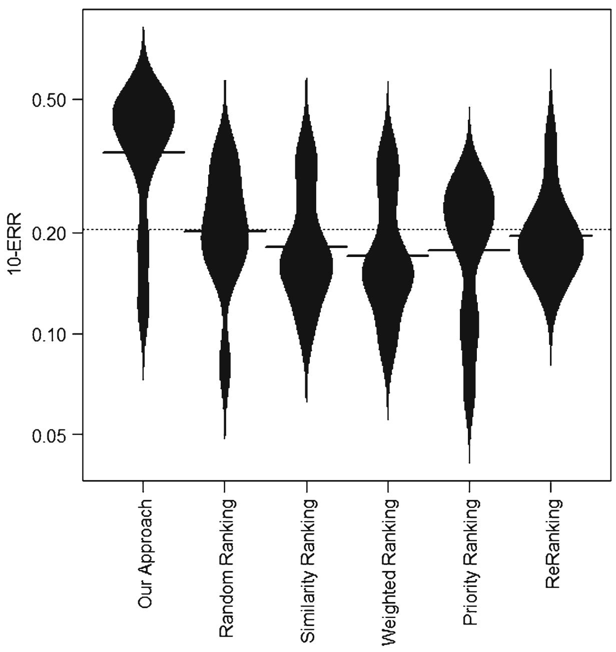


Fig. 10 The result of performance evaluation study between our approach and the existing ranking schemas in terms of 10-ERR

In total, we generate eight alternative ranking schemas. The first alternative ranking schema s^{alt}_{-sim} considers all of the features used in the baseline ranking schema s^{base} except for the similarity feature. Following the same naming convention, the other alternative ranking schemas are denoted by s^{alt}_{-sim}, s^{alt}_{-ctx}, s^{alt}_{-fre}, s^{alt}_{-len}, s^{alt}_{-cmt}, s^{alt}_{-ran}, s^{alt}_{-pro}, and s^{alt}_{-ide} where they consider all the features except for similarity, context, frequency, line length, comment to code ratio, page-rank, probability, and number of identifier respectively. Finally, we compare the performance of the baseline with the alternative ranking schemas to analyze the impact of features on the performance of the baseline approach in terms of k-NDCG and k-ERR, with k ranging from 1 to 20.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data (Sheskin 2007). We conduct Mann-Whitney U test with 5% confidence level

Table 7 Summary of the improvement achieved by our approach comparing with the existing ranking schemas for code example search using 10-NDCG

Ranking schema	10-NDCG	Improvement	p-values	Cliff's delta	Effect size
Our approach	0.46	-	-	-	-
Random ranking	0.26	43.48 %	1.30e-04	0.92	large
Similarity ranking	0.31	32.61 %	2.09e-03	0.78	large
Weighted ranking	0.31	32.61 %	2.09e-03	0.78	large
Priority ranking	0.29	36.96 %	1.05e-03	0.82	large
ReRanking	0.31	32.61 %	4.87e-04	0.86	large

Table 8 Summary of the improvement achieved by our approach comparing with the existing ranking schemas for code example search using 10-ERR

Ranking schema	10-ERR	Improvement	p-values	Cliff's delta	Effect size
Our approach	0.38	-	-	-	-
Random ranking	0.22	42.11 %	1.47e-02	0.64	large
Similarity ranking	0.19	50 %	3.89e-03	0.74	large
Weighted ranking	0.18	52.63 %	2.88e-03	0.76	large
Priority ranking	0.19	50 %	5.20e-03	0.72	large
ReRanking	0.2	47.37 %	8.93e-03	0.68	large

(i.e., p-value < 0.05) to study whether there is a statistical difference between the performance of an alternative schema and the baseline. If the difference of ranking performance between an alternative schema and the baseline is significant, we can conclude that the feature randomized when building the alternative schema is actually important to the success of ranking code examples using learning-to-rank approach.

Result Figures 11 and 12 summarize the result of the performance comparison study between the baseline ranking schema and the alternatives in terms of k-NDCG and k-ERR, with k ranging from 1 to 20. We can see from Figs. 11 and 12 that randomizing similarity, context or frequency obviously reduces the performance of our approach from k-NDCG and k-ERR point of view. Similar to the earlier studies on code example search (McMillan et al. 2013), we look into the comparison result when k is set to 10. Figures 13 and 14 show the 10-NDCG and 10-ERR comparison result between the baseline and alternative ranking schemas. Table 9 lists the decrease in performance and p-values when we compare the baseline schema with the alternative schemas using 10-NDCG and 10-ERR. We observe that randomizing similarity (s^{alt}_{-sim}) or frequency (s^{alt}_{-cntx}) features decreases the performance in terms of 10-NDCG significantly. Randomizing similarity (s^{alt}_{-sim}) or context (s^{alt}_{-cntx}) features decreases the performance in terms of 10-ERR significantly. However, randomizing

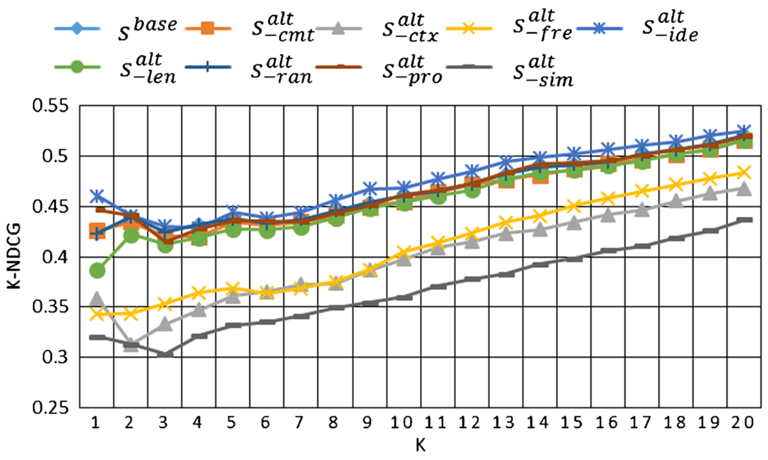


Fig. 11 Studying the impact of features on the performance of our approach based on k-NDCG measure

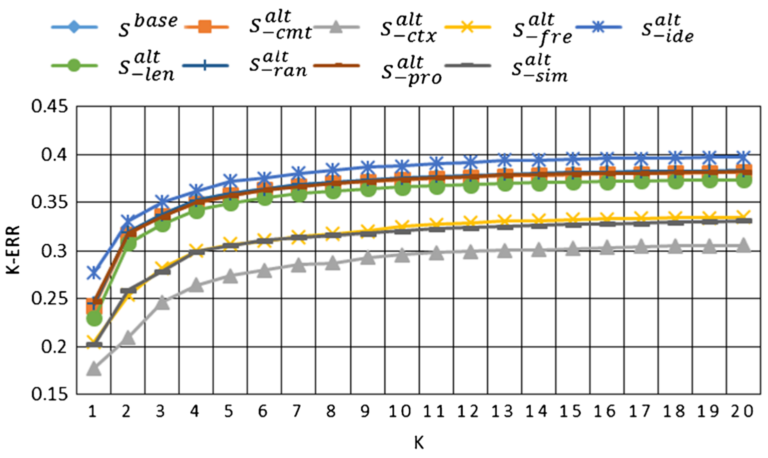


Fig. 12 Studying the impact of features on the performance of our approach based on k-ERR measure

the other features does not affect the performance significantly. Therefore, we can conclude that similarity, context, and frequency are the three influential features for the ranking performance of our approach.

Discussion As illustrated in Figs. 13 and 14, the alternative ranking schema s^{alt}_{-ide} performs better than the baseline ranking schema s^{base}. It indicates that the feature, the number of identifiers per line, constitutes no predictive influence on the code example ranking. To elaborate, assuming two code examples that are relevant to the same query have the same source code, and the statements in one of the code examples are broken down into multiple

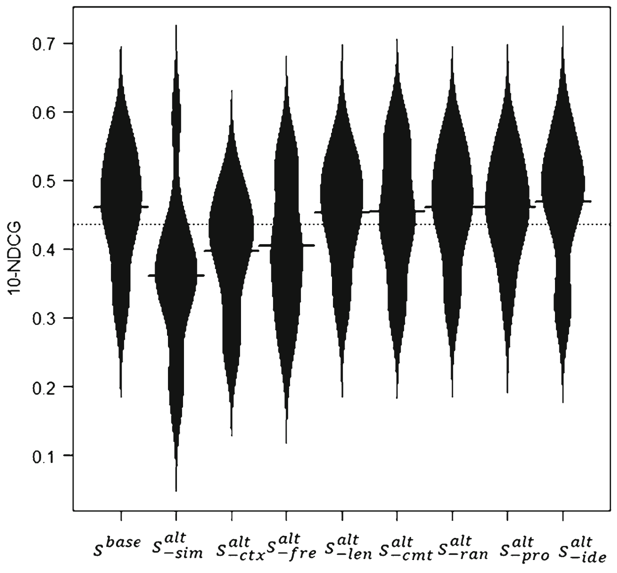


Fig. 13 Performance comparison between baseline and alternative ranking schemas in terms of 10-NDCG measure

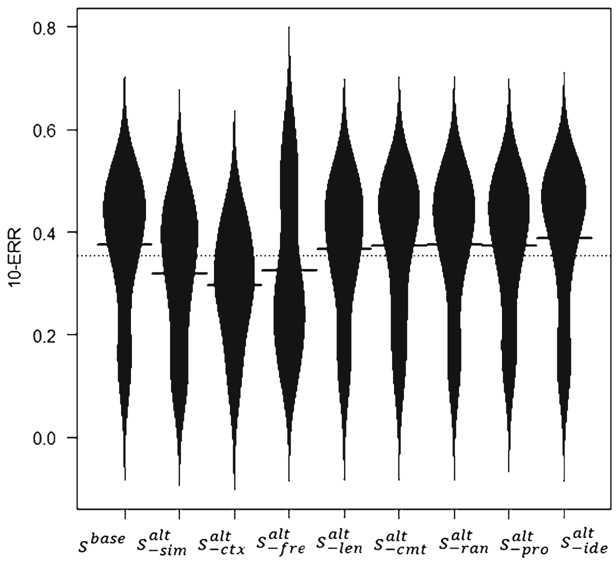


Fig. 14 Performance comparison between baseline and alternative ranking schemas in terms of 10-ERR measure

lines while the other does not.The total number of identifiers of the two code examples are the same since the content of the two code examples are the same. However, the lines of code of the two code examples are different. In other words, the two code examples are different in terms of the number of identifiers per line. However, the code examples should be ranked with the same relevance labels. Therefore, the number of identifiers per line, is not a positive feature to predict the code example ranking.

Similarity, frequency and context are three influential features for the ranking performance of the learning-to-rank approach in the context of code example search and recommendation.

Table 9 Summary of ranking performance for identifying the most influential features

Ranking schemas	10-NDCG			10-ERR		
	Avg.	Impact	p-value	Avg.	Impact	p-value
s^{base} (all features)	0.46	-	-	0.38	-	-
s^{alt}_{-sim}(all features except for similarity)	0.36	−21.74 %	**9.77e-03**	0.32	−15.79 %	**1.37e-02**
s^{alt}_{-ctx}(all features except for context)	0.4	−13.04 %	6.45e-02	0.3	−21.05 %	5.86e-03
s^{alt}_{-fre}(all features except for frequency)	0.4	−13.04 %	2.73e-02	0.32	−15.79 %	0.1055
s^{alt}_{-len}(all features except for line length)	0.45	−2.17 %	0.2807	0.37	−2.63 %	0.2807
s^{alt}_{-cnt}(all features except for comment to ratio)	0.45	−2.17 %	0.6356	0.37	−2.63 %	0.9397
s^{alt}_{ran}(all features except for page-rank)	0.46	−0.00 %	1.00	0.38	−0.00 %	1.00
s^{alt}_{-pro}(all features except for probability)	0.46	−0.00 %	0.6356	0.37	−0.00 %	0.6356
s^{alt}_{-ide}(all features except for identifier)	0.47	+2.17 %	0.5566	0.39	+2.63 %	0.6953

RQ3: Does our approach using the learning-to-rank technique outperform Codota?

Motivation Code examples can help developers learn how to use a specific API or class (Keivanloo et al. 2014). Successful code example search engine should place effective code examples at the top of ranked list to improve the user experience in searching for code examples (Niu et al. 2012). In this research question, we compare our approach with Codota in terms of recommending effective code examples to evaluate the usefulness of our approach.

Approach Similar to the earlier studies on the comparison of Web search engines (Bailey et al. 2007), we design a user study to identify which code search engine is more successful in providing effective code examples at the top of the ranked result set. There are five assessors participating in our user study by expressing the preference between the code examples recommended by our approach and Codota for 50 randomly selected queries. The five assessors are the same people who judge the relevances between the queries and the corresponding code examples in the labeling process. To avoid the issue of knowledge transfer, we assign an assessor with the queries he or she has never judged before in the labeling process. For each query, an assessor is provided with two lists of top 5 answers from our approach and Codota respectively. The two result sets are presented side by side anonymously (Thomas and Hawking 2006). Therefore, the assessors do not know which list belong to which engine. Also for each query, the location (i.e., left or right side of the window) of the result for each engine is selected randomly as suggested by Bailey et al. (2007). Assessors have to express the preference between the two code examples at rank k where $1 \leq k \leq 5$, i.e., 5 code example pairs for one query. Each assessor evaluates 50 code example pairs for 10 queries. In total 250 pairs of code examples are evaluated by five assessors for 50 randomly selected queries as described in Section 4.1.1. If one of the result sets has more preferred code examples than the other one, we refer to it as the winner result set. Finally, we report (1) the average number of preferred code examples in the result set for each query by each engine, and (2) the number of winning result sets for each engine.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data (Sheskin 2007). We conduct Mann-Whitney U test to check if the difference in the numbers of the preferred code examples in the result sets recommended for a query by our approach and Codota is significant. However, Mann-Whitney U test is not suitable to be used to compare the difference in the number of winning results since the recorded observations are boolean. We use Chi-Square test (Greenwood and Nikulin 1996) to determine if there is a significant difference between the studied code search engines in terms of the number of winning result sets. We use the 5 % confidence level (i.e., p-value < 0.05) to identify the significance of the comparison results.

Results Figure 15 shows the average number of preferred answers in the result set recommended by our approach and Codota. We observe that out of five answers per query, our approach achieves 3 preferred code examples while 2 answers of Codota are preferred in the top 5 answers. The p-values for the comparison of preferred code examples between our approach and Codota is $1.72e-04$. In addition, 36 out of 50 queries, our approach can recommend winning result sets. It is 44 % higher than Codota's 14 winning queries. The comparison between Codota and our approach in terms of the number of winning result sets is significant, with the p-value of $1.813e-11$. Therefore, we conclude that our approach can outperform Codota for recommending effective code examples for queries related to Android application development.

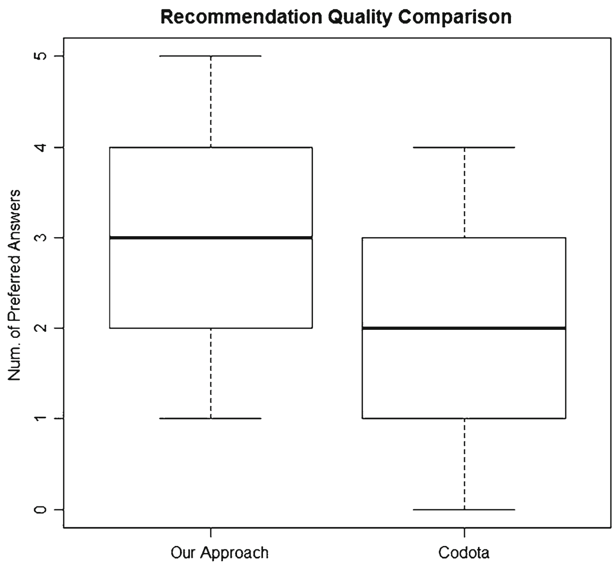


Fig. 15 Comparison results between our learning-to-rank approach and Codota in terms of the number of preferred candidate code examples for one query

Our learning-to-rank approach performs better than Codota by 44% in recommending effective code examples for queries.

6 Threats to Validity

In this section, we analyze the threats to validity for this study following the common guidelines provided in Robert (2002).

Threats to Construct Validity concern whether the setup and measurement in the study reflect real-world situations. In our study, the construct validity threats mainly come from the manual labeling of the relevances between queries and candidate code examples. The subjective knowledge about programming may affect the accuracy in judging on relevance. To reduce the subjectivity in relevance labeling, the relevance between a query and each candidate code example are judged by three independent assessors, and we apply the majority rule if there is a disagreement. In addition, the candidate code examples are shown to the assessors query by query. It might be concerned that an assessor may just select one code example as relevant one and label the other code examples irrelevant. To alleviate the concern, we ask the assessors to independently judge each code example based on the description and the relevance label specification for a specific query. The concern can be further reduced by comparing the relevance labels from three independent assessors and the final validation for randomly selected labels.

In addition, the extraction process of relevant code examples for queries is a keyword-matching process. Therefore, in theory, our code search approach can accept the queries composed of multiple keywords by matching the code snippets with the multiple keywords.

But we have not tested it by ourselves. We will consider evaluating the performance of our approach in the multiple keywords search in our future work.

Threats to Internal Validity concern the uncontrolled factors that may affect the experiment result. In our experiment, the main threats to internal validity come from feature extraction from code snippets. We use Eclipse JDT AST parser to parse the source code of Android projects and extract code snippets. For each code snippet, we compute the values of its features using the definition described in Section 3.2.

Threats to Conclusion Validity concern the relation between the treatment and the outcome. We conducted 10-fold cross validation as sensitivity study. When computing the significance of comparison results, we have used non-parametric test that does not assume the distribution of the data. The difference in the code corpus used by our approach and Codota may affect the conclusion. However, our code corpus is large-scale and created by extracting code snippets from open-source Android projects in GoogleCode. Therefore, it is comparable to the corpus used by Codota.

Threats to External Validity concern whether our experimental results can be generalized for other situations. The main threats to external validity in our study is the representativeness of our corpus and queries. Our code base contains more than 360,000 code snippets extracted from open-source Android projects in Google Code. And we extract 2,500 code examples relevant to 50 queries to create the training data. The size of the training data is sufficient to learn a ranking schema that performs better than other baseline ranking schemas. In practice, the training data can be obtained from the usage logs of code search engines which contains the searching queries and the selected(relevant) candidate code examples. Therefore, the trained ranking schema can perform better using more training data available in practice. As for the queries, the size of our query set (i.e., 50) is comparable to the similar studies on learning-to-rank (Niu et al. 2012), and meets the minimum number of queries recommended for search engine evaluation (Manning et al. 2008).

Threats to reliability validity concern the possibility to replicate this study. All the necessary details that are needed to replicate our work are publicly available: https://www.dropbox.com/s/4gxs85dii1nms9t/Replication.rar?dl=0.

7 Related Work

In this section, we review the studies related to ranking code examples for source code search, and the application of learning-to-rank techniques in other areas.

7.1 Ranking for Source Code Search

Code search engines, such as Google Code Search and Ohloh Code can return code snippets by ranking them based on their textual similarity to queries. Sourcerer (Bajracharya et al. 2006) is a search engine for open-source code. It implements a basic notion of CodeRank by extracting structural information from the code to enable structure-based search instead of conventional keyword-based forms. Thummalapenta and Xie (2007) develop an approach called PARSEWeb which recommends method-invocation sequences (MIS) to benefit object transformation tasks. The approach uses the frequency and length of MISs to rank final result list. McMillan Ye et al. (Grechanik et al. 2012) propose a search engine called Exemplar to find highly relevant software projects to natural-language query. In this search engine, they adopt three different ranking schemes called WOS (word occurrences schema), DCS (dataflow connection schema) and RAS (relevant API calls schema) to sort the list of the retrieved applications. Mishne et al. (2012) present an approach to answer semantic code search. In this approach, returned code snippets are ranked based on the number of similar code snippets which follows the call sequence extracted from the code snippets. Keivanloo et al. (2014) propose a pattern-based approach for spotting working code examples. The approach considers the code snippet's similarity to query, popularity and line length to rank working examples.

The above approaches use a single feature or propose some heuristics to combine the adopted features to rank software-related entities (e.g., code example or software packages). As opposed to the existing work on ranking for code search, we automatically build a ranking schema using machine learning techniques in this paper. In RQ1, we identified the major approaches proposed in the earlier work on ranking code examples, and compared them with our approach.

7.2 Learning-to-Rank

Learning-to-rank has been applied to specific problems related to software maintenance and evolution. Zhou and Zhang (2012) has proposed a method, BugSim, which uses learning-to-rank approach to automatically retrieve duplicate bug reports. The evaluation results show that their proposed method outperforms previous methods using SVM and BM25Fext. Xuan and Monperrus (2014) propose Multric, which locates fault position in source code by applying learning-to-rank. It is observed that Multric performs more effectively than existing approaches in fault localization. Ye et al. (2014) introduce a learning-to-rank approach to rank source code files that might cause a specific bug. It shows that their proposed approach significantly outperforms two state-of-the-art methods in recommending relevant files for bug reports. Binkley and Lawrie (2014) explores the application of learning-to-rank in feature location and traceability. It demonstrates that learning-to-rank works well in the context of software engineering.

Based on the results of applying learning-to-rank in different areas, we can see that learning-to-rank is robust and is widely applicable. In this study, we apply learning-to-rank on code example search for Android application development.

8 Conclusion

In this paper, we have proposed a code example search approach which applies a learning-to-rank technique to automatically build a ranking schema. To create training and testing data, we have identified 12 features of code examples. Then we learn a ranking schema from a training dataset. The learned ranking schema can be used to rank candidate code examples for new queries.

We evaluate our approach using k-NDCG and k-ERR measures. The result of the case study shows that the performance of our approach is significantly better than the existing ranking schemas in searching code examples for Android application development. Among the selected 12 features, similarity, frequency and context are the three influential features for the ranking performance of our approach. Finally, we compare our approach with

Codota, a commercial online code example search engine for Android application development. Our approach outperforms Codota by 44 % in terms of recommending effective code examples.

The better performance of our approach using the learning-to-rank technique can help code search engines place effective code examples at the top of result list and then further improve the user experience in searching for code examples. In the future, we plan to evaluate the performance of learning-to-rank approach using more queries selected from the usage logs of existing code search engines and queries containing multiple keywords. We also would like to study the application of the approach for recommending other types of software artifact, e.g., components or libraries, for pragmatic reuse.

Acknowledgments Many thanks to Liam Gordon, Graem Daly, Bipin Upadhyaya, Ehsan Salamati, and Feng Zhang for their valuable help in relevance labeling at this work.

References

Bailey P, Thomas P, Hawking D (2007) Does brandname influence percerived search result quality? yahoo!, google, and webkumara, Proceedings of ADCS

Bajracharya S, Ngo T, Linstead E, Rigor P, Dou Y, Baldi P, Lopes C (2006) Sourcerer: A search engine for open source code supporting structure-based search. In: Proceedings of International Conference on Object-Oriented Programming Systems, Systems, Languages, and Applications

Bajracharya SK, Ossher J, Lopes CV (2010) Leveraging usages similarity for effective retrieval of examples in code repositories. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pp 157–166

Binkley D, Lawrie D (2014) Learning to rank improves ir in se. In: Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution

Brandt J, Guo P, Lewenstein J, Dontcheva M, Klemmer S (2009) Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp 1589–1598

Breiman L (2001) Random forests p 5–32

Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. In: Proceedings of 7th International World-Wide Web Conference

Bruch M, Schfer T (2008) On evaluating recommender systems for api usages. In: Proceedings of the 2008 international workshop on Recommendation systems for software engineering, p16–20

Burges C, Shaked T, Renshaw E, Lazier A, Deeds M, Hamilton N, Hullender G (2005) Learning to rank using gradient descent. In: Proceedings of the 22nd international conference on Machine learning, p 89–96

Buse RPL, Weimer W (2010) Learning a metric for code readability. IEEE Trans Softw. Eng. 36:546–558

Buse RPL, Weimer W (2012) Synthesizing api usage examples. In: 34th International Conference on Software Engineering

Campbell M, Swinscow TDV (2009) Statistics at square one

Cao Z, Qin T, Liu TY, Tsai MF, Li H (2007) Learning to rank: from pairwise approach to listwise approach. In: Proceedings of the 24th international conference on Machine learning, p 129–136

Chapelle O, Metzler D, Zhang Y, Grinspan P (2009) Expected reciprocal rank for graded relevance. In: Proceedings of the 18th ACM conference on Information and knowledge management, p 621–630

Cliff N (1993) Dominance statistics: Ordinal analysis to answer ordinal questions

Crammer K, Singer Y (2001) Pranking with ranking. In: Advances in Neural Information Processing Systems 14, p 641–647

Freund Y, Iyer R, Schapire RE, Singer Y (2003) An efficient boosting algorithm for combining preferences. In: The Journal of Machine Learning Research, p 933–969

Gallardo-Valencia RE, Sim SE (2009) Internet-scale code search. In: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, p 49–52

Grahne G, Zhu J (2003) Efficiently using prefix-trees in mining frequent itemsets. In: IEEE ICDM Worshop on Frequent Itemset Mining Implementations

Grechanik M, Fu C, Xie Q, McMillan C, Poshyvanyk D, Cumby C (2012) Exemplar: A source code search engine for finding highly relevant applications. IEEE Trans Softw Eng 38:1069–1087

Greenwood PE, Nikulin MS (1996) A guide to chi-squared testing

Harrington P (2012) Machine learning in action

Herbrich R, Graepel T, Obermayer K (2000) Large margin rank boundaries for ordinal regression. In: Advances in Large Margin Classifiers, pp 115–132. MIT Press

Holmes R, Cottrell R, Walker RJ, Denzinger J (2009) The end-to-end use of source code examples: An exploratory study. In: 25th IEEE International Conference on Software Maintenance

Holmes R, Walker RJ (2005) Murphy: Strathcona example recommendation tool. In: Proceedings of European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p 237–240

Holmes R, Walker RJ (2012) Systematizing pragmatic software reuse. ACM Trans Softw Eng Methodol 21

Jaccard P (1901) Tude comparative de la distribution florale dans une portion des alpes et des jura. In: Bulletin de la Socit Vaudoise des Sciences Naturelles 37, p 547–579

Jarvelin K, Kekalainen J (2002) Cumulated gain-based evaluation of ir techniques. In: ACM Transactions on Information Systems, p 422–446

Kapser C, Godfrey MW (2006) Cloning considered harmful" considered harmful. In: 13th Working Conference on Reverse Engineering, p 19–28

Keivanloo I, Rilling J, Zou Y (2014) Spotting working code examples. In: Proceedings of the 36th International Conference on Software Engineering, p 664–675

Kim J, Lee S, Hwang S, Kim S (2010) Towards an intelligent code search engine. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence

Lange BM, Moher TG (1989) Some strategies of reuse in an object-oriented programming environment. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, p 69–73

Li H (2011) A short introduction to learning to rank. IEICE Trans Inf Syst 94:1854–1862

Li P, Burges C, Wu Q (2008) Mcrank: Learning to rank using multiple classification and gradient boosting. In: Proceedings of the 21st Annual Conference on Neural Information Processing Systems, p 897–904

Liu TY (2009) Learning to rank for information retrieval. Found Trends Inf Retr 38:225–331

Lohar S, Amornborvornwong S, AZ, Huang JC (2013) Improving trace accuracy through data-driven configuration and composition of tracing features. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, p 378–388

Manning CD, Raghavan P, Schtze H (2008) Introduction to information retrieval

Manning CD, Raghavan P, Schutze H (2008) Scoring, term weighting, and the vector space model

McMillan C, Poshyvanyk D, Grechanik M, Xie Q, Fu C (2013) Portfolio: Searching for relevant functions and their usages in millions of lines of code. ACM Trans Softw Eng Methodol 22

Mishne A, Shoham S, Yahav E (2012) Typestate-based semantic code search over partial programs. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, p 997–1016

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of 27th international conference on Software engineering, p 284–292

Niu S, Guo J, Lan Y, Cheng X (2012) Top-k learning to rank: labeling, ranking and evaluation. In: Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval, p 751–760

Panichella A, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, Lucia AD (2013) How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on SOftware Engineering (ICSE), pp 522–531

Reiss SP (2009) Semantics-based code search. In: Proceedings of the 31st International Conference on Software Engineering, p 243–253

Robert KY (2002) Design and methods

Robillard MP (2011) A field study of api learning obstacles. Empir Soft Eng 16

Romano J, Kromrey J, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In: AIR Forum, p 1–33

Salton G, Wong A, Yang CS (1975) A vector space model for automatic indexing. In: Communications of the ACM, pp 613–620

Sheskin DJ (2007) Handbook of Parametric and Nonparametric Statistical Procedures. Chapman and Hall/CRC

Shihab E, Zhen M, Ibrahim W. M., Adams B, Hassan A. E (2010) Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In: Proceedings of 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement

Sim S, Gallardo-Valencia R, Philip K, Umarji M, M.A. Lopes C. (2012). In: Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, pp 1361–1370

Stylos J, Faulring A, Yang Z, Myers BA (2009) Improving api documentation using api usage information. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp 119–126

Thomas P, Hawking D (2006) Evaluation by comparing result sets in context. In: Proceedings of ACM International Conference on Information and Knowledge Management

Thummalapenta S, Xie T (2007) Parseweb: A programmer assistant for reusing open source code on the web. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp 204–213

Wang J, Dang Y, Zhang H, Chen K, Xie T, Zhang D (2013) Mining succinct and high-coverage api usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Sotware Repositories, pp 319–328

Xu J, Li H (2007) Adarank: a boosting algorithm for information retrieval. In: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, p 391–398

Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: Proceedings of 30th International Conference on Software Maintenance and Evolution

Ye X, Bunescu R, Liu C (2012) On the naturalness of software. In: Proceedings of IEEE International Conference on Software Engineering

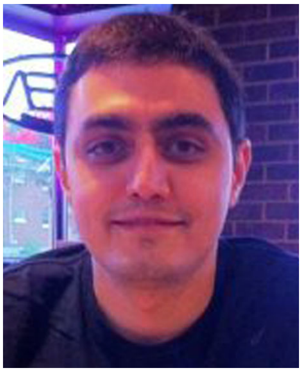
Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 689–699

Ying ATT, Robillard MP (2014) Selection and presentation practices for code example summarization. In: Proceedings of 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering

Zhong H, Xie T, Pei P, Mei H (2009) Mapo: Mining and recommending api usage patterns. In: Proceedings of Euuropean Conference on Object-Oriented Programming, pp 318–343

Zhou J, Zhang H (2012) Learning to rank duplicate bug reports. In: Proceedings of the 21st ACM international conference on Information and knowledge management, pp 852–861

Haoran Niu received the BEng degree from Harbin Institute of Technology, China, in 2013. She is currently working towards the M.A.Sc degree in the Department of Electrical and Computer Engineering at Queen's University, Canada. Her research interests are code search, code recommendation,and their applications in mobile application development.



Iman Keivanloo received his PhD degree in 2013 from Concordia University, Canada. He is currently a Post Doctoral Fellow in the Department of Electrical and Computer Engineering at Queen's University. His research interests include source code similarity search, clone detection, source code recommendation, and their applications for software evolution and maintenance.

Ying Zou is a Canada Research Chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software re-engineering, software reverse engineering, software maintenance, and service-oriented architecture.