Leveraging Official Content and Social Context to Recommend Software Documentation

Jing Li, Zhenchang Xing and Muhammad Ashad Kabir, *Member, IEEE*

Abstract—For an unfamiliar Application Programming Interface (API), software developers often access the official documentation to learn its usage, and post questions related to this API on social question and answering (Q&A) sites to seek solutions. The official software documentation often captures the information about functionality and parameters, but lacks detailed descriptions in different usage scenarios. On the contrary, the discussions about APIs on social Q&A sites provide enriching usages. Moreover, existing code search engines and information retrieval systems cannot effectively return relevant software documentation when the issued query does not contain code snippets or API-like terms. In this paper, we present CnCxL2R, a software documentation recommendation strategy incorporating the content of official documentation and the social context on Q&A into a learning-to-rank schema. In the proposed strategy, the content, local context and global context of documentation are considered to select candidate documents. Then four types of features are extracted to learn a ranking model. We conduct a large-scale automatic evaluation on Java documentation recommendation. The results show that CnCxL2R achieves state-of-the-art performance over the eight baseline models. We also compare the CnCxL2R with Google search. The results show that CnCxL2R can recommend more relevant software documentation, and can effectively capture the semantic between the high-level intent in developers' queries and the low-level implementation in software documentation.

Index Terms—Software documentation, recommendation systems, question and answering sites, ranking model

1 Introduction

With the emergence of Web 2.0 in modern software development, nowadays the behavior of developers is changed in relation to how they look for knowledge to fulfill their information needs [1], [2], as there are large number of accessible official software documentation (*e.g.*, Application Programming Interface (API) documentation[1], language tutorials[2], and language specification[3]) and social media resources (*e.g.*, question and answering sites, personal blogs and technique forums). The official resources provide important information about functionality, structure and parameters for APIs [3], [4]. The social resources provide enriching context in which the developers learn, preserve and share knowledge about software development and maintenance [5]–[7].

Official documentation is an important resource for developers to learn appropriate ways to use an unfamiliar API [8], [9]. Some studies had investigated API learning obstacles and found that there is a mismatch between the needs of consumers and the knowledge provided in software documentation [8], [10]. Robillard *et al.* [8] found that *boilerplate member-level documentation will often not answer the query, and waste developer's time*. Treude *et al.* [11] developed a technique to extract development tasks to navigate official documentation because of the frustration of documentation structure, format and presentation. In addition, many recent studies [9], [12], [13] have targeted the recovery of traceability links between API and its learning resources. Most of existing works recover the traceability via a specific pair, such as <*API, section fragment*> pair [13], [14] and <*API, keyword*> pair [15]. These approaches often take advantage of the content and the context in software documentation but not the advantage of the context in social resources [10], [11]. On the contrary, recent work [15] only takes into account social resources but not the content of software documentation. Developers often issue queries using natural language [15], [16]. However, these approaches mainly accept API-like terms as input query and do not provide any support for natural language query [4], [17], [18]. Thus, the responses generated by them are very low quality.

Some studies [5], [6], [19] show that the developers always benefit from social media such as online question and answering sites when they encountered programming problems. One of the most popular social media for this purpose is Stack Overflow [4], which is an important question and answering venue for developers sharing knowledge on software development. Treude *et al.* [10] proposed a technique to extract insight sentences from Stack Overflow for augmenting API documentation, but do not support natural language query. Campos *et al.* [20] used the Stack Overflow data to recommend question-answer pairs as the solutions for API usage tasks instead of recommending software documentation. Although the Stack Overflow allows users to query in natural language, the users must need to manually check the tedious results to obtain the desired discussion

- *J. Li is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: jli030@e.ntu.edu.sg*
- *Z. Xing is with the College of Engineering and Computer Science, Australian National University, Australia. E-mail: Zhenchang.Xing@anu.edu.au*
- *M.A. Kabir is with the School of Computing and Mathematics, Charles Sturt University, Australia. E-mail: akabir@csu.edu.au*

1. https://docs.oracle.com/javase/8/docs/api/index.html
2. https://docs.oracle.com/javase/tutorial/
3. https://docs.oracle.com/javase/specs/jls/se8/html/index.html
4. https://stackoverflow.com/

Question: Right way to reinitialise transient variable
The Best Answer:
From the docs for ***java.io.Serializable***:
Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void readObjectNoData()
    throws ObjectStreamException;
```

The ***readObject*** method is responsible for reading from the stream and restoring the classes fields. It may call ***in.defaultReadObject*** to invoke the default mechanism for restoring the object's non-static and non-transient fields. The ***defaultReadObject*** method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the ***ObjectOutputStream*** using the ***writeObject*** method or by using the methods for primitive data types supported by ***DataOutput***.
So basically, I think you want:

```
public class MySerializable implements Serializable{
    String value;
    transient String test = "default";
    public MySerializable() {
        test = "init";
    }
    private void readObject(java.io.ObjectInputStream in)
         throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        test = "init";
    }
}
```

Fig. 1. A question and corresponding best answer from Stack Overflow.

threads. For example, the search engine returns 4225 pages with respect to the query "*sort array in Java*" [5]. Even worse, the software documentation related to the query is buried within the returned discussion threads.

When developers search software documentation in natural language, there is a mismatch between the high-level intent in developers' queries and the low-level implementation in software documentation [8]. Moreover, a developer may want to complete a programming task with an unfamiliar API, and he may issue a query that does not contain any words found in the desired software documentation. The traditional information retrieval systems which consider only the textual similarity as the search criterion, cannot return the desired software documentation because the cosine similarity is equal to 0 in the $tf\text{-}idf$ vector space. For example, consider the query "*reinitialise transient variable*", there is no Java API documentation which contains all the three keywords of this query. Thus, we cannot obtain even a single relevant API documentation in Java documentation corpus by this query using traditional retrieval systems. Likewise, the state-of-the-art API usage miner [16] cannot return any relevant API sequence based on the code corpus of Github, but gives prompt of "*Note: your query may not be supported by Java SDK library*"[6]. However, a developer has implemented the task using the class `java.io.Serializable` and methd `readObject` on Stack Overflow[7] as shown in Fig. 1. Therefore, considering the context of APIs on social platforms seems to be more appropriate in a software documentation recommendation strategy.

In this paper, we present CnCxL2R, a recommendation strategy that incorporates the content of software documentation and social context on Stack Overflow into a learning-to-rank schema. Given a natural language query, CnCxL2R recommends software documentation through two key steps: candidate software documentation selection and recommendation by learning to rank. *First*, to bridge the lexical gap between query and documentation, CnCxL2R generates candidate software documents based on not only official content but also social context. *Second*, 22 features are extracted to measure the relevance of a software document to a query. In particular, to further bridge the lexical gap, we utilize a neural language model [16], [21] to embed the context of software documentation on Stack Overflow to capture the semantical similarity between a query and a candidate software document. *Finally*, a ranking model is trained by a learning-to-rank manner to recommend top-k software documents. To evaluate the performance of CnCxL2R, we conducted a large-scale automatic evaluation using the real discussions on Stack Overflow. We also conducted a user study to compare the recommendation performance with Google search. In short, we make the following contributions:

- We propose CnCxL2R, a novel software recommendation strategy incorporating the official content of documentation and social context on Stack Overflow into a learning-to-rank schema. It can perform well with natural language queries comparing with traditional code search.
- We investigate four types of features, *i.e.,* statistical-based, textual, context and popularity features, for learning the software documentation ranker. All the four types of features are easy to derive and enable real-time recommendation.
- We conduct a large-scale automatic evaluation to evaluate the performance of CnCxL2R and investigate the effects of different feature groups and embedding parameters. CnCxL2R achieves the state-of-the-art results against 8 baseline methods.
- We conduct a user study to compare the recommended results by CnCxL2R with Google search. Results show that CnCxL2R significantly outperforms Google search in software documentation retrieval task.

The remainder of this paper is organized as follows. In Section 2, we describe the details of CnCxL2R. In Section 3, we explain the design of experiments. In Section 4, we present the results. Section 5 discusses the threats to validity. After summarizing the related work in Section 6, we conclude the paper in Section 7.

2 Methodology

In this section, we present our approach in details. We first introduce the overall architecture. Then we describe the approach to selecting candidate software documentation. Finally, we present our learning-to-rank approach for software documentation recommendation.

2.1 Overview

Fig. 2 shows the architecture of CnCxL2R, which consists of two core phases: candidate software documentation selection and recommendation by learning-to-rank model.

Our idea is based on the consideration that social context can provide complementary information for official software documentation. In the phase of candidate documentation selection, we take into account three factors: the content of a software document, local and global context of a software document. This process recommends possible software documents for the programming task in a query.

5. https://stackoverflow.com/search?q=sort+array+in+java
6. http://211.249.63.55/
7. http://stackoverflow.com/questions/18893032

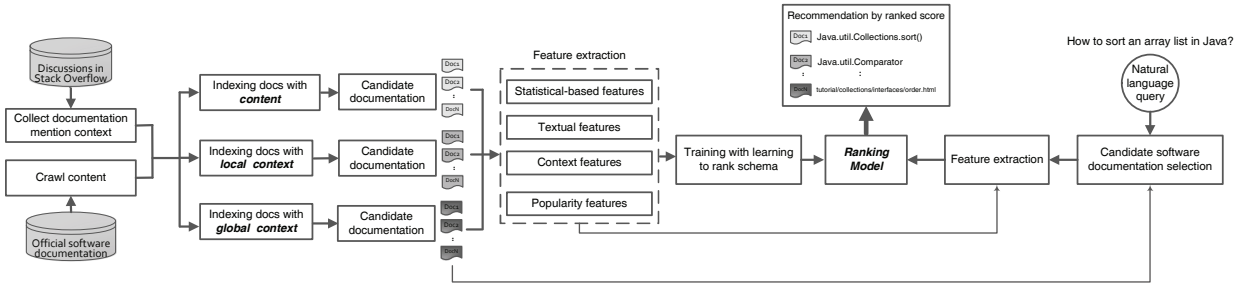

Fig. 2. The architecture of CnCxL2R.

In the phase of recommendation, we train a ranking model using four types of features, *i.e.,* statistical-based, textual, context and popularity features. The selected candidate software documents in the first phase are ranked using the trained model. The input of our approach is a natural language query about programming task and the output is a ranked list of official software documentation. Training the ranking model is an off-line process, while ranking the candidate software documentation is a run-time process that starts when a query is issued by a developer.

2.2 Candidate Software Documentation Selection

Given a query, candidate software documentation selection is a process of selecting a subset of documents related to the programming task in the query. In this paper, software documentation refers to API documentation, language tutorial and language specification. API documentation contains all the information about the functions, classes, return types, arguments and so on. Language tutorials are practical guides for programmers who want to use a programming language to create applications. Language tutorials include hundreds of complete, working examples, and dozens of lessons. Language specification is the definitive technical reference including the semantics of all types, statements, and expressions, as well as threads and binary compatibility. We consider the following three approaches to select the candidate software documents.

2.2.1 Selection by Content

For a query, the content of documentation reflects the surface relevance between the query and software documentation. To retrieve software documents for a query, we need to crawl the content of a corpus of software documentation. We crawl all documents in our corpus (detailed in Section 3.1) and perform following pre-processing.

- There are some code fragments in software documentation, which commonly include class names and method names. We retain code fragments in content because it can provide hints for a query.
- The stopwords are general words and no meaning if they are used alone and appear frequently in text. Thus, we remove the English stopwords provided by the Natural Language Toolkit [22], which contains 127 words (*e.g., all*, *just*, *being*,...).
- Stemming can potentially increase the discriminative power of root words and reduce inflected words to their word stem. We reduce a word to its root using Porter Stemming Algorithm [23].

Given a query, we use Lucene[8] engine and Latent Dirichlet allocation (LDA) [24] model to retrieve candidate software documentation.

Lucene engine: We index each software documentation as an index document in Apache Lucene engine. For each query, we retrieve the top 10 results from corpus using Lucene with BM25 scoring function.

LDA model: We use LDA model to represent the query and the content of software documentation as vectors in topic space. Then we retrieve the top 10 results based on cosine similarity of topic distributions.

2.2.2 Selection by Local Context

Software documentation is mainly written to effectively capture the information about functionality, structure and parameters, but lacking insights about usage scenarios and cautions [10]. Stack Overflow is a popular question and answering site where developers ask programming questions to seek solutions. The content of discussions on Stack Overflow provides enriching context to mine usage scenarios and cautions of APIs [9].

DEFINITION 1 *(Discussion Thread).* A discussion thread consists of a question and all its answers. The question and answers are referred to as posts.

When a software document appears in a discussion thread, its surround texts provide enriching context for its usage scenarios. Now we give the definition of concept of local context for a software document.

DEFINITION 2 *(Local Context).* If a software document is mentioned in a best answer, the texts of the question (title and body) and the best answer are regarded as the local context of the software document.

The definition of local context is based on the consideration that the quality of best answer is better than other answers in the discussion thread. The text of the best answer is the immediate context when a software document appears in a best answer. On the other hand, the question title and question body, describing the programming problem in detail, provide cues to reflect the relevance between the problem and the software documentation in the best answer.

For example, the best answer[9] on Stack Overflow, mentions two API documents: `java.url.regex.Pattern` and `java.util.ArrayList`. This discussion thread consists of a question and three associated answers. In terms of the definition, we only consider question title, question

8. http://lucene.apache.org/
9. http://stackoverflow.com/questions/18625462

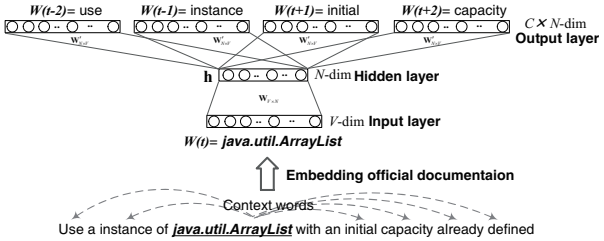

Fig. 3. Training word embedding example with the skip-gram model.

body and the body of the best answer as local context for the two API documents, excluding the other two answers.

We collect all local contexts for the software documents in our corpus. Each local context of a software document is considered as an independent document to be retrieved. Given a query, we use Lucene engine and LDA model to retrieve the most relevant local context and pick out the software documents in the local context as recommendation.

Lucene engine: We index every local context as an index document using Apache Lucene engine. For each query, we retrieve the top 10 software documents from the returned local contexts using Lucene with BM25 scoring function.

LDA model: Query and local context are represented as vectors in topic space. We retrieve the top 10 software documents from the returned local contexts based on cosine similarity of topic distributions.

2.2.3 Selection by Global Context

A specific software document may be mentioned in multiple discussion threads. For example, `java.util.ArrayList` was mentioned 906 times. Thus, there are many local contexts for a specific software document.

DEFINITION 3 ***(Global Context).*** The global context of a software document is the collection of all its local contexts.

For the global context, we use a neural language model for learning term and documentation representations. The key idea is based on Harris' distributional hypothesis [25], which states that words in the same context tends to have similar meanings and similar words have similar vector representations. Traditionally, language models represent each term as a feature vector using one-hot representation, where a vector element that corresponds to the observed word is equal to 1 and 0 otherwise [26]. Recently, neural language models have been proposed to address low-dimensional, distributed embedding of words [27], [28]. These approaches take advantage of neural language models to capture both syntactic and semantic relationships between words. Mikolov's continuous bag-of-words and skip-gram language models [29], [30] are powerful and efficient approaches to learn distributed word embeddings.

Fig. 3 illustrates the training procedure with the skip-gram model when it reaches the current word `java.util.ArrayList`[10]. We define that w_t is the only word on the input layer. N is the hidden layer size. V is the vocabulary size. C is the number of words in the context. $\boldsymbol{x}$

10. http://stackoverflow.com/questions/18625462/get-a-particular-list-of-string-using-regex-in-java/18625626\#18625626

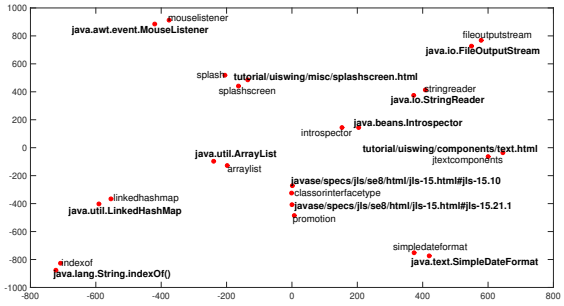

Fig. 4. A 2D projection of embedding natural language words and software documents using PCA.

is the one-hot encoded vector for w_t, which means only one out of V units will be 1 and all other units are 0. The output of hidden layer can be written as

$$\boldsymbol{h} = \boldsymbol{W}^T \boldsymbol{x} = \boldsymbol{V}_{w_t}^T \tag{1}$$

where $\boldsymbol{W}$ is a $V \times N$ input$\rightarrow$ hidden weight matrix. $\boldsymbol{V}_{w_t}$ is the vector representation of the input word w_t.

On the output layer, each output is computed using the hidden$\rightarrow$ output matrix:

$$p\left(w_{c,j} = w_{O,c} | \, w_t\right) = \frac{\exp\left(u_{c,j}\right)}{\sum_{j'=1}^{V} \exp\left(u_{j'}\right)} \tag{2}$$

where w_t is the input word; $w_{c,j}$ is the j-th word on the c-th panel of the output layer; $w_{O,c}$ is the actual c-th word in the output context word. $u_{c,j}$ is the net input of the j-th unit on the c-th panel of the output layer,

$$u_{c,j} = \boldsymbol{V'}_{w_j}^T \cdot \boldsymbol{h}, \; for \; c = 1, 2, ..., C \tag{3}$$

where $\boldsymbol{V'}_{w_j}^T$ is the output vector of the j-th word in the vocabulary, w_j and $\boldsymbol{V'}_{w_j}^T$ is taken from a column of the hidden$\rightarrow$ output weight matrix, $\boldsymbol{W'}$.

When training the skip-gram model to predict C context words, the loss function is written as

$$\begin{aligned} E &= -\log p\left(w_{O,1}, w_{O,2}, ..., w_{O,C} | \, w_t\right) \\ &= -\log \prod_{c=1}^{C} \frac{\exp\left(u_{c,j_c^*}\right)}{\sum_{j'=1}^{V} \exp\left(u_{j'}\right)} \\ &= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} \exp\left(u_{j'}\right) \end{aligned} \tag{4}$$

where j_c^* is the index of the actual c-th output context word in the vocabulary.

We use the skip-gram to learn embeddings of natural language words and software documentation. Fig. 4 illustrates a 2-D projection of vectors of natural language words and software documentation in our dataset using principal component analysis (PCA). In the embedding space, semantically close words are likewise close in the embedding space while they are not close based on surface similarity such as term overlapping and TF-IDF weight. Especially, the vectors of the word and software documentation with same intent have the shortest distance. For example, the word "arraylist" is close to the API documentation `java.util.ArrayList`.

Following [31], we use bag-of-words model to average out the vectors of the individual words in a query. Given a

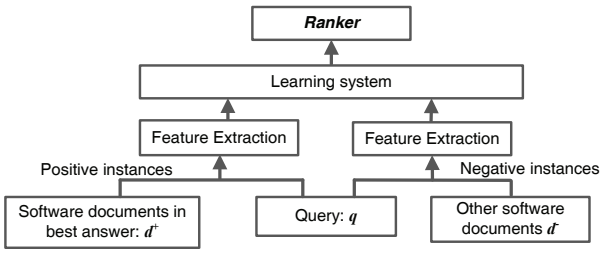

Fig. 5. Learning-to-rank architecture.

query, we retrieve top 10 software documents based on the cosine similarity between the average vector and software documentation vector.

2.3 Recommendation by Learning to Rank

Given a programming query, our task is to recommend a ranked list of software documents that are most relevant to the intent in the query. Mose specifically, we formulate the software documentation recommendation task as a learning-to-rank problem.

2.3.1 Problem Formulation

For the ranking task, a successful approach is to treat it as a supervised machine learning problem [32]. The most typical setup is the supervised learning-to-rank schema, which is as follows: Assume that there is a corpus of documents. In the training process, a set of queries and retrieved lists are provided. Each query is associated with a set of retrieved documents with relevance judgments. In the learning-to-rank schema, each query-document pair is represented by a set of features. The learning-to-rank schema automatically learns the optimal way of combining these features. A set of such pairs is used to train a machine learning algorithm, then a ranking function is built to rank the documents pertaining to the query.

Fig. 5 shows the architecture of learning-to-rank process. $D = \{d_1, d_2, ..., d_n\}$ represents the retrieved candidate software documents for a query q. The candidate set comes with their relevancy judgments, where the software documents in the best answer are annotated as positive instances ($\langle q, d_+ \rangle$) and negative for otherwise ($\langle q, d_- \rangle$). Our goal is to build a ranking model which facilitates optimal ranking of the candidate list D for a query q. More formally, the task is to learn a scoring function $F(q, d)$:

$$F(q,d) = \sum_{k=1}^{K} \omega_i \cdot \phi_i(q,d) \tag{5}$$

where each feature $\phi_i(q, d)$ measures a specific relationship between query q and candidate software documentation d. ω_i is the weight of the i-th feature (total K features), and is learned during the training. The optimization procedure of learnin-to-rank tries to find the scoring function that can rank the relevant software documents at the top of candidate list.

There are mainly three approaches to learn scoring function, namely, pointwise [33]–[35], pairwise [36]–[38] and listwise [39]–[41]. Pointwise approach is the most simple way to build ranking model, which defines the loss function based on individual documentation. The pointwise commonly be viewed as regression approach by minimizing a loss function. Pairwise approach transforms the ranking problem to pairwise classification, taking candidate documentation pairs as instances in learning instead of individual documentation. Listwise approach generates a candidate list through the comparison between two documents. Listwise approach takes lists as instances in learning and loss function is defined on basis. Our approach falls within the category of pairwise.

2.3.2 Feature Extraction

In this section, we describe the details of feature engineering for training the ranking schema in our approach. Totally, we extract 22 features, which fall into four groups: Statistical-based, Textual, Context, and Popularity features as listed in Table 1. Furthermore, we divide the 22 features into 3 categories: Q-D means that the feature is dependent on both query and candidate document, Q represents that the feature is calculated by query regardless of candidate document, D means that the feature only depends on candidate document regardless of query.

Statistical-based Features. We use 10 statistical-based features [21], [42] in our learning-to-rank schema, which are widely used in information retrieval community and shown in Table 1.

We define that q represents a query, which consists of t terms $q_1, q_2, ..., q_t$. The number of occurrences of the query term q_i in document d is denoted as $c(q_i, d)$. Document frequency $df(q_i)$ reflects the number of documents containing q_i in the document collection. $|C|$ is the total number of documents in the document collection and $|d|$ is the length (*i.e.,* the number of terms) of document d. We obtain 10 features (F1-F10) based on the above definitions. Note that F5, F6 and F7 belong to Q category, and the other 7 features belong to Q-D category.

Textual Features. Textual features are the basic features used to judge relevancy between a query and a software document.

textualSim: This feature computes the textual similarity between a query and a software document using cosine similarity.

$$textualSim(V_q, V_d) = \cos(V_q, V_d) = \frac{V_q^T V_d}{\|V_q\| \, \|V_d\|} \tag{6}$$

where V_q is a query vector and V_d is a vector of a software document based on bag-of-words model.

textualBM25: BM25 [43] is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. The BM25 score between query q and a software document d is computed as follows,

$$BM25(q,d) = \sum_{q_i \in q} \frac{idf(q_i) \cdot c(q_i,d) \cdot (k+1)}{c(q_i,d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{avgdl}\right)} \cdot \frac{(k_3+1) \cdot c(q_i,d)}{k_3 + c(q_i,d)} \tag{7}$$

where $avgdl$ is the average software documentation length in the entire document corpus. k_1, k_3 and b are free parameters. Following the benchmark of learning-to-rank system [42], we set $k_1 = 2.5, k_3 = 0$ and $b = 0.8$.

TABLE 1
Learning features.

Group	Feature name	Description	Category
Statistical-based	F1	$\sum_{q_i \in q \cap d} c(q_i, d)$	Q-D
	F2	$\sum_{q_i \in q \cap d} \log(c(q_i, d) + 1)$	Q-D
	F3	$\sum_{q_i \in q \cap d} \frac{c(q_i,d)}{\|d\|}$	Q-D
	F4	$\sum_{q_i \in q \cap d} \log\left(\frac{c(q_i,d)}{\|d\|} + 1\right)$	Q-D
	F5	$\sum_{q_i \in q \cap d} \log\left(\frac{\|C\|}{df(q_i)}\right)$	Q
	F6	$\sum_{q_i \in q \cap d} \log\left(\log\left(\frac{\|C\|}{df(q_i)}\right)\right)$	Q
	F7	$\sum_{q_i \in q \cap d} \log\left(\frac{\|C\|}{c(q_i,C)} + 1\right)$	Q
	F8	$\sum_{q_i \in q \cap d} \log\left(\frac{c(q_i,C)}{\|d\|} \cdot \log\left(\frac{\|C\|}{df(q_i)}\right) + 1\right)$	Q-D
	F9	$\sum_{q_i \in q \cap d} c(q_i, d) \cdot \log\left(\frac{\|C\|}{df(q_i)}\right)$	Q-D
	F10	$\sum_{q_i \in q \cap d} \log\left(\frac{c(q_i,d)}{\|d\|} \cdot \frac{\|C\|}{c(q_i,C)} + 1\right)$	Q-D
Textual	$textualSim$	The textual similarity between a query and a software document based on bag-of-words model.	Q-D
	$BM25$	The BM25 score between a query and a software document.	Q-D
	$isClueInQuery$	Whether the clue word in the URL of a software document is contained in a query.	Q-D
	$isHashmark$	Whether the URL of a software document contains a fragment identifier (hashmark #).	D
	$docCategory$	One of categories (official APIs, official tutorials and official specifications).	D
	$numSlash$	Number of slashes in the URL of a software document.	D
	$lengthUrl$	Length of strings in the URL of a software document.	D
	$lengthDoc$	Number of terms in a software document.	D
	$lengthQuery$	Number of terms in a query.	Q
Context	$contextSim$	The context cosine similarity between a query and a software document.	Q-D
Popularity	$refFrequency$	The number of times that a software document is referenced on Stack Overflow.	D
	$isBest$	It is true when a software document ever occurred in best answers.	D

isClueInQuery: Whether the clue word in the URL of a software document is contained in the query. For example, *isClueInQuery* is true when the clue word "*HashSet*" in the URL *https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html* is contained in the query "*Understanding HashSet*".

isHashmark: Whether the URL of a software document contains a fragment identifier (hashmark #). For example, *isHashmark* is true because the URL *https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getResourceAsStream-java.lang.String-* contains a hashmark #.

docCategory: The official software documentation consists of official APIs, official language tutorials and official language specifications. *docCategory* is equal to 1 when the documentation is an official API, 2 for official tutorial, and 3 for official specification.

numSlash: Number of slashes in the URL of a software document.

lengthUrl: Length of strings in the URL of a software document.

lengthDoc: Number of terms in a software document.

lengthQuery: Number of terms in a query.

Context Features. After training the word2vec corpus, we can get the vector representation of each word. Suppose that the length (*i.e.,* the number of terms) of a query and a software document are N and M, respectively. In order to get the text vector, we average the word vectors in the text. w_q and w_d denote the text vector of a query and a software document, respectively. We can get the context similarity between the query and the software documentation as follows:

$$contextSim(w_q, w_d) = \cos(w_q, w_d) = \frac{\frac{1}{NM} \cdot \sum w_i \cdot \sum w_j}{\left\| \frac{\sum w_i}{N} \right\| \left\| \frac{\sum w_j}{M} \right\|} \quad (8)$$

Popularity Features. Much meta-data is available on Stack Oveflow. We can extract features from these meta-data to measure the popularity of a software document.

refFrequency: This feature measures the number of times that a software document is referenced on Stack Overflow.

isBest: It is true when a software document was occurred in a best answer.

2.3.3 Learning a Ranker

In our approach, we train the ranking model using one of the state-of-the-art learning-to-rank algorithms, namely, LambdaMART [44], a boosted tree version of LambdaRank [45], that won the Yahoo! Learning to Rank Challenge.

In our approach, the input of learning-to-rank algorithm is the training instances which represent each query-documentation pair as a feature vector. d_i and d_j denote candidate software documentation i and j in the given query q, respectively. Based on feature extraction in Section 2.3.2, x_i and x_j are feature vector for $< q, d_i >$ and $< q, d_j >$, respectively. LambdaMART builds a regression tree to model the functional gradient of the cost function of interest, and evaluate at all the training pairs [44], [46]. The λ-gradients can be written as follows:

$$\lambda_{ij} = S_{ij} \left| \Delta NDCG \frac{\partial C_{ij}}{\partial \sigma_{ij}} \right| \quad (9)$$

where $S_{ij} \in \{-1, 1\}$ is equal to 1 if the documentation d_i is more relevant than the documentation d_j, and is equal to -1 if the documentaion d_i is less relevant than the documentation d_j. $\Delta NDCG$ is the NDCG (Normalized Discounted Cumulative Gain) gained by swapping those two software documents. Let $F(x)$ is the ranking function, and $\sigma_{ij} = F(x_i) - F(x_j)$ is the difference in ranking scores for the pair of software documents. $C_{ij} = F(x_j) - F(x_i) + \log\left(1 + e^{F(x_i) - F(x_j)}\right)$ is the cross-entropy cost applied to

the logistic of the difference of the scores. Each point then sums its λ-gradients for all pairs P in which it occurs:

$$\lambda_i = \sum_{j \in P} \lambda_{ij} \tag{10}$$

A positive lambda indicates a push toward the top rank position and a negative lambda indicates a push toward the lower rank positions. Finally, the ranking function $F(x)$ can be learned based on the λ-gradients and Newton-Raphson line step [44].

3 Experimental Setup

In this study, we performed a set of experiments with the data from official software documentation websites and Stack Overflow website. We now describe our experimental setup: data collection, performance measures and comparison baselines.

3.1 Data Collection

There are two types of text data used in this research: the text of software documentation and the text of discussion threads on Stack Overflow.

Corpus for Word Embedding. In this research, we focus on Java software documentation on its official sites and Java discussion threads on Stack Overflow. The discussion threads on Stack Overflow provide enriching insights for software documentation by means of hyperlinks. The official documentation mainly provides information about functionality, structure and parameters instead of insights and feedbacks on Stack Overflow. Usually, a query is expressed in natural language, and much closer to the discussions on Stack Overflow. Thus, we use the discussion threads on Stack Overflow as the corpus for word embedding.

For the questions and answers on Stack Overflow, their quality significantly varies from post to post. In order to guarantee the quality of text, we extract these discussion threads using the datadump archive available on the Stack Exchange website [11] that satisfy the following criteria:

- The question tags contain "Java". This condition filters out the local and global context about "Java" from the sea of information.
- The score of question is larger than 0. This condition guarantees that at least a developer has voted the question as an "useful" question.
- The question has an answer which is accepted as the best answer. Meanwhile, the score of the best answer is larger than 0. This condition guarantees the quality of the best answer.
- The best answer must contain at least one hyperlink which links to official Java API documentation, or official Java tutorial, or official Java specification.

Based on the above criteria, we totally collect $30,272$ discussion threads based on data dump released on August 2015. We randomly select $24,217$ discussion threads (account for 0.8) as training discussion threads and other 6055 threads (account for 0.2) as test discussion threads. For the training discussion threads, we take the following text pre-processing:

- Select question title, question body and best answer body as text document for embedding.
- Remove code snippets and english stop words, and change all characters to lowercase.
- Add the actual URL address after the anchor text of hyperlink of software documentation.
- Normalize the URLs of software documentation of different Java versions. For example, in our vocabulary, we use "*api_docs/api/java/util/ArrayList.html*" to label the three URLs (*https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html*, *https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html*, and *https://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html*).

We use these $24,217$ discussion threads as training corpus for word embedding. To learn word representations, we use the skip-gram model implemented in Gensim [12]. The context window size is set to 10 and the minimal word frequency is 5. Finally, we collect $1,520$ Java software documents from training discussion threads.

Crawl Official Documentation. After training the word embeddings, we get $1,520$ Java software documents and we consider these software documents as our research corpus. We crawl these $1,520$ software documents via the hyperlinks in discussion threads. We take the following text preprocessing:

- Usually, a class or interface level of documentation is very long because it contains thorough method descriptions. For example, `java.util.ArrayList` (*https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html*) contains 31 methods in its html file. In such cases, we only crawl the class or interface description in the beginning of html file as the content of the URL.
- If an URL of a software document contains a fragment identifier (hashmark #), we crawl the content of the section that the hashmark identifies. For example, the URL of *https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#get-int-* mentions the method `get(int index)`. We only crawl the section of `get(int index)` in the html file.

Dataset for Learning A Ranker. In this study, we only focus on the corpus consisting of $1,520$ Java software documents as the demonstration of effectiveness of CnCxL2R. We leave the cold-start problem in the future research. Thus, we select the discussion threads whose best answers contain at least one software document of our corpus to learn a ranker. Based on the above setting, we collect $12,020$ discussion threads for training a learning-to-rank model. In the corpus of test discussion threads, we collect $2,924$ discussion threads for testing our recommendations.

For a selected discussion thread, we assume that the question title contains sufficient information for the problem

11. https://archive.org/details/stackexchange

12. https://radimrehurek.com/gensim/

in the question. We consider the question title as the input query of CnCxL2R. The pair <*query*, *software documentation in the best answer*> is treated as a positive instance and an randomly sampled pair as a negative instance.

3.2 Performance Measures

We choose four performance metrics which are widely used for recommendation evaluation: precision at position k, recall at position k, mean average precision (MAP) and mean reciprocal rank (MRR) [47], [48].

Precision at position k ($P@k$): $P@k$ is a metric to measure top k positions of a ranked list using two levels of relevance judgment (relevant and irrelevant). For a natural language query q, let D_k be the set of top-k recommended official documents and D_g be the set of ground-truth documents. $P@k$ is calculated by $P@k = \frac{|D_k \cap D_g|}{k}$.

Recall at position k ($R@k$): $R@k$ is a measure for evaluating the fraction of top-k documents that are relevant to the query that are successfully retrieved. $R@k$ can be defined as follows: $R@k = \frac{|D_k \cap D_g|}{|D_g|}$.

Mean average precision (MAP): MAP is defined as the mean of average precision (AP) over a set of queries. Let Q be the set of all test queries. The set of relevant documents for $q_j \in Q$ is $\{d_1, ..., d_k, ..., d_{m_j}\}$, and R_{jk} is the set of ranked results from the top results until you get to document d_k, then $MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})$. When a relevant document is not retrieved at all, the precision value in the above equation is taken to be 0.

Mean reciprocal rank(MRR): The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. The MRR is the average of the reciprocal ranks of results for a sample of queries Q: $MRR(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{rank_j}$, where $rank_j$ refers to the rank position of the first relevant document for the j-th query.

3.3 Comparison Baselines

In order to validate the effectiveness of the proposed approach, we evaluated 8 models in our experiments and provided the corresponding results as baselines.

- **CnBM25**: This is the baseline model which selects candidate software documents by content described in Section 2.2.1. Note that this model indexes each candidate with the content stated in itself and retrieves candidates using BM25 [43] scoring function[13].
- **CnLDA**: This baseline is also based on the content of software documentation in Section 2.2.1. This model represents query and content of documentation as vectors in topic space. In our experiments, we use Latent Dirichlet Allocation (LDA) [24] for topic modeling. In particular, we computed the perplexity [24] of a held-out test set of training data to determine the number of topics. The perplexity is monotonically decreasing in the likelihood of the test data, and is algebraicly equivalent to the inverse of the geometric mean per-word likelihood. We choose the number of topics (*i.e.*, 200) with the lowest perplexity.
- **CnDoc2vec:** This baseline [49] is based on the doc2vec approach proposed by Google, which concatenates the paragraph vector with several word vectors from a paragraph and predicts the following word in the given text. The document vectors are learned from the content of software documentation.
- **CxBM25**: This baseline is based on the local context of software documentation described in Section 2.2.2. This model indexes each local context as a document and retrieves candidates using BM25. Given a query, this method is to retrieve the most similar context and find out official software documents in the best answer as the recommendation.
- **CxLDA**: This baseline is similar with **CnLDA**. The difference is that this baseline uses the topic space of local context for query and documentation representation. The lowest perplexity results in 300 topics for LDA models.
- **LoCxDoc2vec:** This baseline is similar with **CnDoc2vec**, but the document vectors are learned from local contexts.
- **GloCxCos**: This baseline is base on the global context describe in Section 2.2.3. This model represents natural language words and software documents as vectors in shared embedding space [29]. The cosine similarity is used for retrieval.
- **GloCxAsy**: This baseline is based the global context and an asymmetric similarity measure [50] is used to retrieve candidates.
- CnCxL2R: This method is our proposed model which takes into account content, local and global context of software documentation in a learning-to-rank schema as shown in Fig. 2.

13. We use the BM25 implementation provided by Apache Lucene.

4 Experimental Results

In this section, we present our experimental results. First, we compare the performance of different models described in Section 3.3 and report the performance of different feature groups. Second, we present the effect of embedding dimensionality. Finally, we present a comparative result against Google Search.

4.1 Performance of Different Models

In this section, we compare our model CnCxAPI with other 8 baseline models. Table 2 shows the recommendation performance by using different evaluation metrics discussed in Section 3.2. We have the following observations:

(1) From Table 2, we note that the performance of $P@k$ of CnCxL2R significantly outperforms all the remaining baseline models for all values of $k(k = 1, 2, 5, 10)$ in our dataset. We also conduct a statistical t-test and the results show that the improvements between the CnCxAPI model and the other 8 baselines are statistically significant ($p < 0.05$). In particular, $P@10$ is very small because it is calculated

TABLE 2
Performance ($P@k$, MAP and MRR) for different models. The bold formate indicates the best performance. † indicates that the difference between the results of *CnCxL2R* and other models are significant with $p < 0.05$ under t-test.

Model	$P@1$	$P@2$	$P@5$	$P@10$	$R@1$	$R@2$	$R@5$	$R@10$	MAP	MRR
CnBM25	0.1334	0.1073	0.0743	0.0510	0.1128	0.1766	0.3021	0.4089	0.2214	0.2248
CnLDA	0.0854	0.0672	0.0435	0.0317	0.0789	0.1028	0.2343	0.3562	0.1637	0.1729
CnDoc2vec	0.1172	0.0879	0.6725	0.0487	0.9216	0.1321	0.2745	0.3698	0.1893	0.2014
LoCxBM25	0.1790	0.1461	0.1018	0.0719	0.1482	0.2368	0.4066	0.5641	0.2969	0.3032
LoCxLDA	0.1225	0.0942	0.0743	0.0519	0.0996	0.1508	0.2946	0.4029	0.2075	0.2107
LoCxDoc2vec	0.1341	0.1012	0.0677	0.0532	0.1084	0.1833	0.3278	0.4237	0.2368	0.2435
GloCxCos	0.1517	0.1228	0.0847	0.0607	0.1276	0.2022	0.3430	0.4817	0.2522	0.2575
GloCxAsy	0.1811	0.1523	0.1087	0.0834	0.1646	0.2573	0.4394	0.6233	0.3153	0.3364
CnCxL2R	**0.2238†**	**0.1766†**	**0.1230†**	**0.0904†**	**0.1842†**	**0.2862†**	**0.4853†**	**0.7043†**	**0.3779†**	**0.3868†**

by $P@10 = \frac{|D_{10} \cap D_g|}{10}$, where D_{10} is the set of top-10 recommended software documents and D_g is the set of ground-truth documents. For most of discussion threads on Stack Overflow, the best answers commonly contain only one software document, so $|D_{10} \cap D_g| = 1$ in most cases. Thus, the ideal value of $P@10$ is slightly above 0.1. Although $P@10$ of CnCxL2R is very small (*i.e.,* 0.0904), it almost reaches the ideal value.

(2) The performance of $R@k$ of CnCxL2R significantly outperforms the other 8 baselines for all values of $k(k = 1, 2, 5, 10)$. It is worth noting that CnCxL2R achieves highest recall (0.7043) at $k = 10$.

(3) We can also observe that the performance of MAP and MRR of CnCxL2R significantly outperform all other methods. The MAP of CnCxL2R is equal to 0.3779, which achieves 70.69%, 130.84%, 99.63%, 27.28%, 82.12%, 59.58%, 49.84%, and 20.54% relative improvements over *CnBM25*, *CnLDA*, *CnDoc2vec*, *LoCxBM25*, *LoCxLDA*, *LoCxDoc2vec*, *GloCxCos* and *GloCxAsy*, respectively. Likewise, the performance of MRR of CnCxAPI achieves relative improvements with 72.06%, 123.71%, 92.05%, 27.57%, 83.57%, 58.85%, 50.21%, and 14.98%, respectively.

(4) The context-based model outperforms corresponding content-based model (*e.g., LoCxBM25* better than *CnBM25*). This observation confirms that there exists an information gap between the intent in queries and the implementation in software documentation.

(5) Another observation is that BM25-based models (*CnBM25* and *LoCxBM25*) achieve better performance than topic-based models (*CnLDA* and *LoCxLDA*). The reason for this difference may be that the length of query is short while the length of software documentation is often long. In such case, the query is not effectively represented in topic space. Thus it leads to poor performance when retrieving software documents using distributed topics of a query.

4.2 Performance of Different Feature Combinations

As shown in Table 1, we divide the 22 features into four groups and three categories. In this Section, we investigate the performance of different feature combinations in the learning-to-rank framework.

As shown in Table 3, we totally produce eight feature combinations in terms of feature groups and categories. The first combination considers all of the 22 features in the learning-to-rank schema. The combinations (*i.e.,* #2 – #8) consider all features excluding context features, textual features, statistical-based features, popularity features, Q-class features, D-class features and QD-class features, respectively. The results show that the improvements between the first combination and the other seven combinations are statistically significant at $p < 0.05$, in terms of MAP, MRR, $P@1$, $P@10$, $R@1$ and $R@10$. These results confirm that these features derived from official documentation and social context, are effective in recommending software documentation for developers.

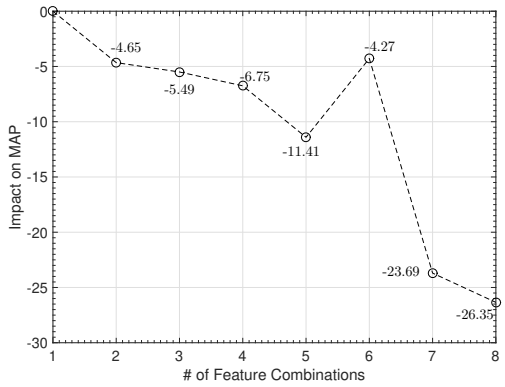

Fig. 6. The impact of feature combinations on MAP.

We consider the first combination (*i.e.,* #1 in Fig. 6) as the baseline. Then we define the performance impact as $\frac{P_n - P_1}{P_1} \times 100\%$, where P_1 is the performance metric of the first combination, and P_n $(n = 2, ..., 8)$ is the performance metric from the second to eighth combination. Fig. 6 shows the impact of different feature combinations on MAP. We can observe that the second feature combination (excluding context features) decreases the performance of MAP by -4.65% compared with the first combination (all features), and followed by -5.49%, -6.75%, -11.41%, -4.27%, -23.69%, and -26.35% for the feature combination #2 to #8, respectively. Among the combinations of #2, #3, #4, and #5, the performance of #5 (excluding popularity features) is worst with impact of -11.41%. The sixth combination (excluding query-dependent features) has little effect with impact of -4.27%. On the contrary, the seventh combination (excluding document-dependent features) and eighth combination (excluding query-document dependent features) have influential effect on MAP with impact of -23.69% and -26.35%, respectively.

4.3 Effect of Embedding Dimensionality

The dimensionality of word embedding is an important factor for embedding performance. Furthermore, the

TABLE 3
Performance of different feature combinations. The bold formate indicates the best performance. † indicates that the difference between the results of the first combination and other combinations are significant with $p < 0.05$ under t-test.

#	Feature Combination	MAP	MRR	P@1	P@10	R@1	R@10
1	all features	**0.3779†**	**0.3868†**	**0.2238†**	**0.0904†**	**0.1842†**	**0.7043†**
2	all features excluding context features	0.3611	0.3763	0.2104	0.0814	0.1739	0.6110
3	all features excluding textual features	0.3582	0.3747	0.2089	0.0886	0.1705	0.6868
4	all features excluding statistical-based features	0.3540	0.3700	0.2044	0.0895	0.1669	0.6950
5	all features excluding popularity features	0.3392	0.3533	0.1981	0.0803	0.1674	0.6281
6	all features excluding Q-class features	0.3624	0.3791	0.2172	0.0799	0.1792	0.6905
7	all features excluding D-class features	0.3055	0.3202	0.1700	0.0747	0.1394	0.5854
8	all features excluding QD-class features	0.2991	0.3098	0.1374	0.0854	0.1124	0.6565

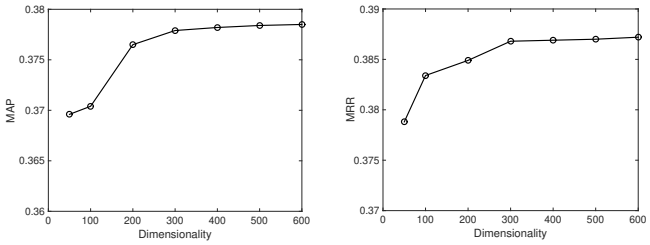

Fig. 7. The effect of different dimensionality of word embedding for learning-to-rank.

performance of word embedding directly affects candidate documentation selection and the context similarity feature listed in Table 1. Thus, the word embedding can affect the final ranking performance in many ways. When we train the word representations, we need specify the parameter of number of dimensionality. In this Section, we investigate the effect of embedding dimensionality with values 50, 100, 200, 300, 400, 500, and 600.

Fig. 7 shows the effect of different dimensionality on learning-to-rank schema in terms of MAP and MRR. We observe that the increase of embedding dimensionality from 50 to 300, leads to a gradual improvement in performance. The optimal performance can be achieved at the dimensionality of 300. After 300 dimensions, the performance is not improved significantly, while the training procedure becomes extremely slow. Thus in this research, we set 300 as the value of the dimensionality.

4.4 Comparison with Google Search

Given a natural language query, the search engine of Stack Overflow can only return a list of discussion threads and could not directly retrieve software documentation. However, Google search can return a list of software documentation. Moreover, Google search is the most popular help-seeking tool when develops need to search for solutions for programming problems. Thus, we compare the results of CnCxL2R with the Google search engine.

4.4.1 Setup

Among the 2924 testing discussion threads, we randomly select 30 discussion threads and consider the titles of these discussion threads as queries. Note that the 30 queries are not used in the training set and are listed in Table 4. Given a natural language query, we first get a list of web pages via Google search engine. As we focus on the official documentation in this research, we restrict the retrieved results through appending official sites in the queries. For example, the first query in Table 4 was performed with "*get objects from a BlockingQueue site:docs.oracle.com/javase*" using Google search engine in August, 2016. For each query in Table 4, we get the top-10 software documents by the Google search engine and CnCxL2R separately.

We use three metrics [16], [51] to measure the performance of the two list of software documents. The metric FR represents the rank of the first relevant software documentation in the results list of a query. This metric is important as most users scan the results from top to bottom. The smaller the number of FR, the better the performance. The metric $RR5$ denotes the relevancy ration of software documents in the top 5 results. The metric $RR10$ represents the relevancy ration of software documents in the top 10 results. $RR5$ and $RR10$ range from 0 to 100. The higher the values of $RR5$ and $RR10$, the better the performance.

We recruited two developers who both are postdoctoral fellows with 4-years+ programming experiences in Java, to manually check the two lists generated by Google search engine and CnCxL2R. Each software documentation in candidate list was marked relevant or irrelevant, indicating whether the developer considered this software documentation is related to the task in the query. The two developers separately annotated the top 10 software documents for each evaluated query and then we combine the results. If any inconsistency found while combining the results, the developers discussed among them and then remarked until a consensus is reached. The Cohen's kappa coefficient is equal to 0.846, which shows that the two annotators have a good agreement amongst themselves.

4.4.2 Comparison Results

Table 4 shows the performance comparison of Google search and CnCxL2R. In particular, the symbol "-" in second column indicates that there is no relevant software documentation returned by Google search engine for the query. The last row shows the average performance in terms of the defined three metrics.

Compared to FR of Google search, CnCxL2R achieves better performance with average FR of 1.3. For most of the queries (23 out of 30 queries), CnCxL2R is able to recommend relevant software documentation at the first position in the result list. On the contrary, Google search is able to handle only 11 queries in which the first returned software documentation is relevant to the given query. Specifically, for 5 out of 30 queries, Google search could not return any relevant software documentation in top 10 results. The p-value of FR comparison shows that there is a statistical significance of the improvement of CnCxL2R over

TABLE 4
Comparison between Google search and CnCxL2R (FR: the rank of the first relevant documentation. $RR5$: top 5 relevancy ratio. $RR10$: top 10 relevancy ratio. "java.xxx.xxx" indicates Java API documentation. "specs_xxx/xxx" indicates Java language specification. "tutorial/xxx/xxx" indicates Java tutorial. † indicates the differences between CnCxL2R and Google are significant with $p < 0.05$ under t-test.)

query	Google Search			CnCxAPI			Top 3 relevant documents by CnCxL2R
	FR	$RR5$	$RR10$	FR	$RR5$	$RR10$	
get objects from a BlockingQueue	1	20	60	1	40	80	java.util.concurrent.BlockingQueue; java.util.concurrent.ArrayBlockingQueue; java.util.concurrent.ConcurrentLinkedQueue
java string split inconsistency	6	0	20	1	60	70	java.lang.String.split(); java.lang.String.substring(); java.lang.String.contains()
create a Series using Iterators	2	40	20	2	60	40	java.util.Iterator; java.util.Iterable; java.util.ListIterator
rules governing narrowing of double to int	1	20	30	1	60	60	specs_5.1.3. Narrowing Primitive Conversion; specs_3.10.2. Floating-Point Literals; specs_15.25. Conditional Operator ? :
add submenu to MenuItem	1	80	70	2	40	20	tutorial/uiswing/components/dialog.html; tutorial/uiswing/components/menu.html; tutorial/uiswing/layout/visual.html
show a timer in a Java Frame	1	40	30	1	60	50	tutorial/uiswing/misc/timer.html; java.util.Timer; javax.swing.Timer
create utility to cast objects to beans	5	20	10	4	40	40	java.lang.Class.cast(); java.lang.Object.getClass(); java.lang.Class.getResourceAsStream()
java Client/Server App will not readLine()	-	0	0	1	60	80	java.io.BufferedReader.readLine(); java.io.InputStream.read(); tutorial/networking/sockets/clientServer.html
synchronize concurrent collections	1	20	10	1	100	90	java.util.concurrent.ConcurrentHashMap; java.util.concurrent.ConcurrentLinkedQueue; java.util.Collections.synchronizedList()
pull double out of string with matcher	-	0	0	1	80	70	java.util.regex.Matcher; java.util.regex.Pattern; java.lang.String.matches()
java hashcode() collision for objects	-	0	0	1	60	50	java.lang.Object.hashCode(); java.lang.String.hashCode(); java.lang.System.identityHashCode()
java reverse string	1	60	50	2	60	40	java.lang.StringBuilder; java.lang.String.format(); java.lang.String.compareTo()
get char from multi-string array	3	40	30	1	100	60	java.lang.String.charAt(); java.lang.String.split(); java.lang.String.replace()
search for the index of an element in an array	1	40	30	1	80	50	java.util.Arrays.binarySearch(); tutorial/java/nutsandbolts/arrays.html; java.util.ArrayList.get()
replce last space in a string	2	40	20	1	100	80	java.lang.String.trim(); java.lang.String.split(); java.lang.String.indexOf()
different font for combo box in Java	4	20	10	1	60	40	tutorial/uiswing/components/combobox.html#renderer; java.awt.Font; tutorial/uiswing/components/combobox.html
add days to date in Java	4	20	30	1	100	70	java.text.SimpleDateFormat; api.java.util.Date; java.text.DateFormat
support println in a class	1	20	10	1	100	70	java.io.PrintStream.println(); api.java.lang.Object; java.lang.System.setOut()
remove row from jtable	1	40	30	1	80	70	javax.swing.table.DefaultTableModel; javax.swing.JTable; javax.swing.table.AbstractTableModel
primitive cast and assignments in Java	2	40	30	1	60	60	spec_5.2. Assignment Contexts; spec_3.10.2. Floating-Point Literals; spec_5.5. Casting Contexts
draw points as ovals using java swing	1	60	60	1	100	80	tutorial/uiswing/painting/index.html; javax.swing.JComponent.paintComponent(); tutorial/uiswing/concurrency/dispatch.html
julian date to regular date conversion	2	20	10	1	60	80	java.text.SimpleDateFormat; api.java.util.Date; java.text.DateFormat
reinitialise transient variable	-	0	0	3	40	60	java.io.Serializable; specs_15.28. Constant Expressions; java.util.Scanner
invalid class exception: no valid constructor	5	20	20	1	40	50	java.io.Serializable; java.lang.Class.newInstance(); java.lang.IllegalArgumentException
combine explicit locks with synchronized methods	1	100	50	1	80	70	java.util.concurrent.locks.ReentrantLock; java.util.concurrent.locks.Lock; tutorial/essential/concurrency/locksync.html
parse a number from a string	2	40	40	3	40	70	java.text.SimpleDateFormat; java.lang.Integer.parseInt(); java.text.NumberFormat.parse()
format double with zeros on left and right side	-	0	0	1	60	70	java.text.DecimalFormat; tutorial/uiswing/layout/visual.html; api.java.util.Formatter
wait for a key to be pressed inside loop	4	20	20	1	60	60	tutorial/uiswing/misc/keybinding.html; tutorial/uiswing/concurrency/dispatch.html; tutorial/uiswing/events/keylistener.html
get single bytes from multi-byte variable in java	3	20	20	1	40	50	java.nio.ByteBuffer; java.lang.String.String(); spec_5.2. Assignment Contexts
change PWD of linux from JSP	-	0	0	2	40	60	java.lang.Runtime.exec(); java.lang.ProcessBuilder.directory(); java.util.ResourceBundle
average	> 2.3	28	24	1.3†	65†	61†	

Google search. Compared to the Google search (average $RR5$ is 28 and average $RR10$ is 24), CnCxL2R achieves much higher average value for $RR5$ (65) and $RR10$ (61). Therefore, CnCxL2R can recommend much more relevant software documents in top 10 results than Google search. The differences between these two approaches in terms of $RR5$ and $RR10$ are statistically significant at $p < 0.05$.

The last column shows the top three relevant documents recommended by CnCxL2R, which consist of Java API documents, Java language specifications and Java tutorials. We observe that CnCxL2R can more effectively respond to bug-like queries over Google search. For instance, Google search could not return relevant API documentation given the query "*java Client/Server App will not readLine()*", but CnCxL2R can recommend eight relevant API documents in top 10 results. More importantly, the first recommended software documentation (`java.io.BufferedReader.readLine()`) is extremely relevant to the query. Likewise, CnCxL2R can recommend five relevant software documents whereas the Google search can only return one document with respect to the exception query "*invalid class exception: no valid constructor*".

Another observation is that CnCxL2R can effectively respond to those queries which do not contain API-like words. For example, the query "*format double with zeros on left and right side*" does not explicitly contain API-like words. Thus, Google search cannot effectively handle this case and fails to return any relevant software documents. However, CnCxL2R can recommend seven high quality software documents including `java.text.DecimalFormat` and `api.java.util.Formatter`. In the same way, for the query "*reinitialise transient variable*" and "*change PWD of linux from JSP*", CnCxL2R can recommend extremely relevant software documents `java.io.Serializable` and `java.lang.Runtime.exec()`, respectively. In such cases, CnCxL2R can bridge the lexical gap and identify the semantics between the query and the documentation.

However, CnCxL2R cannot recommend good software documentation for some queries. For example, for the query "*create utility to cast objects to beans*", CnCxL2R cannot recommend any documentation about "*beans*". After manually checking our training dataset, we find that there is not enough context about the topic "*beans*". This causes the coverage of this query incomplete. In the future, we will collect more context of different topics and improve the coverage of our training dataset accordingly.

5 Threats to Validity

The automatically annotated data is one of the threats to validity of our experiment. We built the ground-truth dataset via judging software documents in the best answers in Section 4.1. For some cases, the recommended software documents which are not in the best answer, may be useful for the issued query. It was a tedious task to annotate the ground truth of the $14,944$ discussion threads in our corpus. However, our human evaluation provides a supplementary study about this threat. The experiment setting of automatic evaluation makes the $P@k$ very low, but CnCxL2R almost reaches the ideal value. Even so, the high values of $R@k$ indicate the validity of CnCxL2R.

In Section 4.4, we employed two annotators to label the two recommended lists generated by Google search and CnCxL2R. This is a threat to validity because the judgments are based on the knowledge background of the two annotators. For a natural language query about programming

task, there are many approaches to implement the task with different APIs. To make it more reliable, we require the two annotators to discuss until consensus is reached when a judgment is inconsistent for a query. The relevant software documents reflect the most common implementations for the programming task in a query. We believe that the discussion can eliminate the threat to some extent.

Another threat is the training data. In our approach, the local context and global context are from the discussion threads on Stack Overflow. Most of the contexts are free-form texts posted by the users of Stack Overflow, rather than the quotation of official documentation. For the free-form texts posted by users, there may be spelling mistakes and typos. However, the quality of the best answers is better than other answers on Stack Overflow [52]. Thus, we restrict the context in best answers and we believe it can eliminate the threat to some extent.

Last but not least, the coverage of the training data is a threat to validity. In this study, we only test the performance of CnCxL2R on Java documentation and collect the discussion threads about Java topic on Stack Overflow. The coverage of the training dataset potentially affects the query response performance. In this study, the performance on Java related programming task is reasonable. It is still uncertain how CnCxL2R will perform on other datasets. In future, we will perform more evaluations on different training sets and create a better training set.

6 Related Work

In this Section, we present a literature review on mining software documentation, divided into API usage, API recommendation and harnessing Stack Overflow data.

6.1 API usage and API Recommendation

There is a number of techniques which mine API usage [53]–[55] and recommend APIs from the perspectives of the content of documentation and code corpus.

API usage patterns are frequent API method call sequences. Xie *et al.* [53] presented MAPO, which mined API usages from open source repositories based on existing source code search engines. Given a query consisting of method names or class names, MAPO can generate a list of method call sequences. Wang *et al.* [54] developed API usage pattern miner (UP-Miner) to mine succinct and high-coverage API usage patterns from source code. Given a user-specified API method, UP-Miner returns code snippets as reuse candidates via mining frequent closed API-method invocation sequences. Acharya *et al.* [56] developed a framework to automatically extract frequent partial orders among user-specified APIs, assisting effective API reuse and checking. Moritz *et al.* [57] presented Export, which can automatically mine and visualize API usages in large source code repositories. Given a starting API, Export can recommend complex API usage examples from a large repository.

Recently, Pham *et al.* [58] proposed an approach to learn API usages from bytecode of Android mobile applications, which uses Hidden Markov Model to represent method call sequences. Raghothaman *et al.* [51] presented SWIM to capture API usage patterns based on the defined call sequences. SWIM is based on open-source code repositories and click-through data from the Bing search engine. Gu *et al.* [16] developed DeepAPI, a deep learning approach to capture API usage sequences using the Recurrent Neural Network encoder-decoder models. Compared to SWIM, DeepAPI can recommend more accurate API usage sequences.

TABLE 5
Comparison with other related approaches. Column *Information* specifies that which information is used in each approach (Source Code (SC), Content of Documentation (CN), Context on Stack Overflow (CX). Column *Input Type* specifies the input format for each approach. Column *Output Type* specifies the output format for each approach.

Approach	Information	Input Type	Output Type
MAPO [53]	SC	API name	API usage
UP-Miner [54]	SC	API name	API usage
ExPort [57]	SC	API name	API usage
SWIM [51]	SC	API-like query	API usage
DeepAPI [16]	SC	API-like query	API usage
XSnippet [62]	SC	Code snippet	Code snippet
Baker [4]	CN	Code snippet	Software documentation
Krec [17]	CN	Code snippet	Software documentation
TaskNavigator [11]	CN	Semi-free-form	Section fragment
ROSF [63]	SC	Free-form	Code snippet
Portfolio [59]	SC	Free-form	API usage
RACK [15]	CX	Free-form	API class
CnCxL2R (our)	CN, CX	Free-form	Software documentation

McMillan *et al.* [59] developed Portfolio to recommend and visualize relevant APIs and their usages for code search. Their experiments show that Portfolio can find more relevant APIs compared to Google Code Search. Given simple text phrases, Chan *et al.* [60] proposed subgraph search approach for API recommendation via modeling API invocations as an API graph. Thung *et al.* [61] developed a technique which recommends API methods for a query of textual description of a feature request.

Recently, Subramanian *et al.* [4] developed a constraint-based approach to identify fine-grained type references, method calls, and field references in source code snippets. This approach can correctly link source code examples to official API documentation. Robillard *et al.* [17] developed a technique to automatically detect knowledge items and extract word patterns in software documentation. They use these patterns to recommend reference API documentation for code fragments. Rahman *et al.* [15] proposed RACK, using regular expressions to extract API class and to build keyword-API associations from the posts on Stack Overflow. Given some code search keywords, RACK can recommend a list of relevant API classes. This work is close to our work, but it is based on Java class level instead of Java method level. Moreover, it did not exploit the content of official documentation instead of using keyword-API co-occurrence of social context.

Different from the existing work from the view points of program analysis and mining code repositories, our approach exploits the official content of documentation and high-quality social context on Stack Overflow for recommendation. A relative comparison of our approach with the existing approaches is presented in Table 5.

6.2 Harnessing Stack Overflow Data

The Stack Overflow data has attracted much research interest [64]–[66] in recent years. Some tools were developed for assisting software development. Bacchelli *et al.* [67]

developed Seahawk, an Eclipse plugin that can automatically integrate crowd knowledge of Stack Overflow into the Integrated Development Environment (IDE). This tool brings the convenience for developers that they can directly access Stack Overflow data without switching their work context. Ponzanelli *et al.* [68] presented Prompter, a self-confident recommender system that automatically searches and identifies relevant Stack Overflow discussions under the code context in the IDE. San Pedro *et al.* [69] proposed RankSLDA, recommending questions for collaborative Q&A systems based on developers' topics of expertise. Cordeiro *et al.* [70] developed a tool, recommending question answering web resources in IDE based on the information of exception stack traces. Treude *et al.* [10] presented SISE, automatically augmenting API documentation with "insight sentences" from Stack Overflow.

In addition, some researchers have contributed their efforts for program comprehension using Stack Overflow data. The study [71] revealed that their tool and Stack Overflow data are capable of sometimes coming up with surprising insights that aid a developer both for program comprehension and software development. Treude *et al.* [72] investigated how developers ask and answer questions on the Web. Linares-Vásquez *et al.* [73] took an empirical study on how do API changes trigger Stack Overflow discussions. Recently, Nadi *et al.* [3] performed an empirical investigation into the obstacles developers face while using the Java cryptography APIs based on 100 StackOverflow posts, 100 GitHub repositories, and survey input from 48 developers.

In contrast to these work, the goal of our approach is to bridge the information gap between natural language query and official software documentation harnessing Stack Overflow data.

7 Conclusion and Future Work

Traditional code search engines and existing API recommendation systems often do not perform well with natural language queries, especially which do not contain API-like terms. In this paper, we attempt to leverage official content and social context to recommend software documentation. Our proposed solution, named CnCxL2R, exploits the official content and social context in a learning-to-rank schema. We identify 22 features to learn a ranker. We conduct a set of experiments to evaluate the effectiveness of our approach. The results of these experiments show that CnCxL2R outperforms 8 baseline models and is effective for natural language queries.

As future work, we plan to create a better training set with higher coverage for improving the performance of recommendation. As human feature engineering consumes massive manpower, we will also investigate the automatic feature engineering in our approach using deep learning.

References

[1] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. ICSE*, 2007, pp. 344–353.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[3] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: why do java developers struggle with cryptography apis?" in *Proc. ICSE*, 2016, pp. 935–946.

[4] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proc. ICSE*, 2014, pp. 643–652.

[5] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empir. Softw. Eng.*, vol. 19, no. 3, pp. 619–654, 2014.

[6] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec, "Discovering value from community activity on focused question answering sites: a case study of stack overflow," in *Proc. KDD*, 2012, pp. 850–858.

[7] B. Vasilescu, V. Filkov, and A. Serebrenik, "Stackoverflow and github: Associations between software development and crowdsourced knowledge," in *Proc. SocialCom*, 2013, pp. 188–195.

[8] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empir. Softw. Eng.*, vol. 16, no. 6, pp. 703–732, 2011.

[9] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *Proc. ICSE*, 2012, pp. 47–57.

[10] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proc. ICSE*, 2016, pp. 392–403.

[11] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Trans. Softw. Eng*, vol. 41, no. 6, pp. 565–581, 2015.

[12] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proc. ICSE*, 2013, pp. 832–841.

[13] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proc. ICSE*, 2015, pp. 869–879.

[14] H. Jiang, J. Zhang, X. Li, Z. Ren, and D. Lo, "A more accurate model for finding tutorial segments explaining apis," in *Proc. SANER*, vol. 1, 2016, pp. 157–167.

[15] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *Proc. SANER*, vol. 1, 2016, pp. 349–359.

[16] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," 2016.

[17] M. P. Robillard and Y. B. Chhetri, "Recommending reference api documentation," *Empir. Softw. Eng.*, vol. 20, no. 6, pp. 1558–1586, 2015.

[18] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empir. Softw. Eng.*, pp. 1–33, 2016.

[19] J. Li, L. Bao, Z. Xing, X. Wang, and B. Zhou, "Bpminer: mining developers' behavior patterns from screen-captured task videos," in *Proc. SAC*. ACM, 2016, pp. 1371–1377.

[20] E. C. Campos, L. B. de Souza, and M. d. A. Maia, "Searching crowd knowledge to recommend solutions for api usage tasks," pp. 1–32, 2016.

[21] G. Zhou, Y. Zhou, T. He, and W. Wu, "Learning semantic representation with neural networks for community question answering retrieval," *Knowl.-Based Syst.*, vol. 93, pp. 75–83, 2016.

[22] S. Bird, "Nltk: the natural language toolkit," in *Proc. COLING/ACL*, 2006, pp. 69–72.

[23] P. Willett, "The porter stemming algorithm: then and now," *Program*, vol. 40, no. 3, pp. 219–223, 2006.

[24] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. Jan, pp. 993–1022, 2003.

[25] Z. S. Harris, "Distributional structure," in *Pap. struct. transform. linguist.*, 1970, pp. 775–794.

[26] M. Grbovic, N. Djuric, V. Radosavljevic, F. Silvestri, and N. Bhamidipati, "Context-and content-aware embeddings for query rewriting in sponsored search," in *Proc. SIGIR*, 2015, pp. 383–392.

[27] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[28] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: a simple and general method for semi-supervised learning," in *Proc. ACL*, 2010, pp. 384–394.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR*, 2013.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. NIPS*, 2013, pp. 3111–3119.

[31] T. Van Nguyen, A. T. Nguyen, and T. N. Nguyen, "Characterizing api elements in software documentation with vector representation," in *Proc. ICSE*, 2016, pp. 749–751.
[32] D. H. Dalip, M. A. Gonçalves, M. Cristo, and P. Calado, "Exploiting user feedback to learn to rank answers in q&a forums: a case study with stack overflow," in *Proc. SIGIR*, 2013, pp. 543–552.
[33] P. Li, Q. Wu, and C. J. Burges, "Mcrank: Learning to rank using multiple classification and gradient boosting," in *Proc. NIPS*, 2007, pp. 897–904.
[34] L. Li and H.-T. Lin, "Ordinal regression by extended binary classification," in *Proc. NIPS*, 2006, pp. 865–872.
[35] W. S. Cooper, F. C. Gey, and D. P. Dabney, "Probabilistic retrieval based on staged logistic regression," in *Proc. SIGIR*, 1992, pp. 198–210.
[36] M.-F. Tsai, T.-Y. Liu, T. Qin, H.-H. Chen, and W.-Y. Ma, "Frank: a ranking method with fidelity loss," in *Proc. SIGIR*, 2007, pp. 383–390.
[37] T. Pahikkala, E. Tsivtsivadze, A. Airola, J. Boberg, and T. Salakoski, "Learning to rank with pairwise regularized least-squares," in *Proc. SIGIR*, vol. 80, 2007, pp. 27–33.
[38] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun, "A general boosting method and its application to learning ranking functions for web search," in *Proc. NIPS*, 2008, pp. 1697–1704.
[39] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *Proc. ICML*, 2007, pp. 129–136.
[40] M. N. Volkovs and R. S. Zemel, "Boltzrank: learning to maximize expected ranking gain," in *Proc. ICML*, 2009, pp. 1089–1096.
[41] H. Valizadegan, R. Jin, R. Zhang, and J. Mao, "Learning to rank by optimizing ndcg measure," in *Proc. NIPS*, 2009, pp. 1883–1891.
[42] T. Qin, T.-Y. Liu, J. Xu, and H. Li, "Letor: A benchmark collection for research on learning to rank for information retrieval," *J. Inf. Retr.*, vol. 13, no. 4, pp. 346–374, 2010.
[43] S. E. Robertson, S. Walker, S. Jones *et al.*, "Okapi at trec-3," in *Proc. TREC*, 1994, pp. 109–126.
[44] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao, "Adapting boosting for information retrieval measures," *J. Inf. Retr.*, vol. 13, no. 3, pp. 254–270, 2010.
[45] C. Quoc and V. Le, "Learning to rank with nonsmooth cost functions," *Proc. NIPS*, vol. 19, pp. 193–200, 2007.
[46] C. J. Burges, "From ranknet to lambdarank to lambdamart: An overview," *Learning*, vol. 11, pp. 23–581, 2010.
[47] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*, 1999, vol. 463.
[48] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Trans. Inf. Syst.*, vol. 20, no. 4, pp. 422–446, 2002.
[49] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. ICML*, 2014, pp. 1188–1196.
[50] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. ICSE*, 2016, pp. 404–415.
[51] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean - code search and idiomatic snippet synthesis," in *Proc. ICSE*, May 2016, pp. 357–367.
[52] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne, "Finding high-quality content in social media," in *Proc. WSDM*, 2008, pp. 183–194.
[53] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proc. MSR*, 2006, pp. 54–57.
[54] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proc. MSR*, 2013, pp. 319–328.
[55] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *Proc. ESEM*. IEEE, 2013, pp. 5–14.
[56] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proc. FSE*, 2007, pp. 25–34.
[57] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *Proc. ASE*, 2013, pp. 646–651.
[58] H. V. Pham, P. M. Vu, T. T. Nguyen *et al.*, "Learning api usages from bytecode: a statistical approach," in *Proc. ICSE*, 2016, pp. 416–427.
[59] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proc. ICSE*, 2011, pp. 111–120.
[60] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proc. FSE*, 2012, p. 10.
[61] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of api methods from feature requests," in *Proc. ASE*, 2013, pp. 290–300.
[62] N. Sahavechaphan and K. Claypool, "Xsnippet: mining for sample code," *ACM Sigplan Notices*, vol. 41, no. 10, pp. 413–430, 2006.
[63] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, "Rosf: Leveraging information retrieval and supervised learning for recommending code snippets," *IEEE Trans. Serv. Comput.*, vol. 10.1109/TSC.2016.2592909, 2016.
[64] J. Li, Z. Xing, D. Ye, and X. Zhao, "From discussion to wisdom: web resource recommendation for hyperlinks in stack overflow," in *Proc. SAC*. ACM, 2016, pp. 1127–1133.
[65] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, "Answering questions about unanswered questions of stack overflow," in *Proc. MSR*, 2013, pp. 97–100.
[66] J. Zhang, H. Jiang, Z. Ren, and X. Chen, "Recommending apis for api related questions in stack overflow," *IEEE Access*, vol. 6, pp. 6205–6219, 2018.
[67] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *Proc. RSSE*, 2012, pp. 26–30.
[68] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proc. MSR*, 2014, pp. 102–111.
[69] J. San Pedro and A. Karatzoglou, "Question recommendation for collaborative question answering systems with rankslda," in *Proc. RecSys*, 2014, pp. 193–200.
[70] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proc. RSSE*, 2012, pp. 85–89.
[71] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *Proc. CSMR*, 2013, pp. 57–66.
[72] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?: Nier track," in *Proc. ICSE*, 2011, pp. 804–807.
[73] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proc. ICPC*, 2014, pp. 83–94.

```
@article{TSCLiXK21,
author    = {Jing Li and
Zhenchang Xing and
Muhammad Ashad Kabir},
title     = {Leveraging Official Content and Social Context to Recommend Software
Documentation},
journal   = {{IEEE} Trans. Serv. Comput.},
volume    = {14},
number    = {2},
pages     = {472--486},
year      = {2021},
url       = {https://doi.org/10.1109/TSC.2018.2812729},
doi       = {10.1109/TSC.2018.2812729},
}
```