SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

Context-Aware API Recommendation Using Tensor Factorization

Yu Zhou[1,3], Chen Chen[1], Yongchao Wang[1], Tingting Han[2] & Taolue Chen[2,3*]

[1]*Nanjing University of Aeronautics and Astronautics, Nanjing* 211106*, China;*

[2]*Birkbeck, University of London, London, UK;*

[3]*State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China*

Abstract An activity constantly engaged by most programmers in coding is to search for appropriate application programming interfaces (APIs). Contextual information is widely recognized to play a crucial role in effective API recommendation, but it is largely overlooked in practice. In this paper, we propose **c**ontext-aware **A**PI **r**ecommendation using **t**ensor **f**actorization (CARTF), a novel API recommendation approach in considering programmers' working context. To this end, we use tensors to explicitly represent the query-API-context triadic relation. When a new query is made, CARTF harnesses word embeddings to retrieve similar user queries, based on which a third-order tensor is constructed. CARTE then applies non-negative tensor factorization to complete missing values in the tensor and the Smith–Waterman algorithm to identify the most matched context. Finally, the ranking of the candidate APIs can be derived based on which API sequences are recommended. Our evaluation confirms the effectiveness of CARTF for class-level and method-level API recommendations, outperforming state-of-the-art baseline approaches against a number of performance metrics, including success rate, precision, and recall.

Keywords API recommendation, Tensor factorization, Context awareness, Word embedding, Intelligent software development

Citation Y. Zhou et al. Context-Aware API Recommendation Using Tensor Factorization. Sci China Inf Sci, for review

1 Introduction

Application programming interfaces (APIs) are encapsulated reusable software libraries, which are essential building blocks of large scale software systems. To enable efficient software development, a variety

* Corresponding author (email: t.chen@bbk.ac.uk)

of API recommendation approaches have been proposed to, for instance, recommend API sequences [1] or API-related documents [2] for specific programming tasks [3]. In general, these API recommendation approaches can, based on the inputs, be classified into two categories: recommendation with or without explicit queries. The former recommendation methods require carefully designed queries to capture programmers' intentions. Typically, the problem is framed as an information retrieval task. Queries are transformed into, e.g., word vectors, and then textual matching is conducted to identify the most matched APIs [4]. To overcome the lexical gap between natural languages and code, additional artifacts are usually leveraged. These artifacts may include API documentation [5], API invocation graphs [6], library usage patterns [7], code surfing behaviors of the developers and API invocation chains [8], and posts in Q&A websites [9, 10]. For the latter category, because there are no explicit queries as input, context information is needed to infer the programmers' intention. Typically, they may include surrounding code snippets [11–13], API usage graph [14], or even ambient projects [15].

In practice, when programmers make queries, some part of the code is already available, so they are looking for appropriate APIs that are consistent with the existing code snippet. As a motivating example, imagine that a programmer is implementing a method to iterate a hashmap in Java. Listing 1 depicts the place where the developer gets stuck. In this case, the programmer may formulate the query "hashmap key iteration." Multiple candidate API sequences could be returned by an API recommender, examples of which are shown in Listing 2 and Listing 3. Although both candidates are relevant to the query, if '`java.util.Map.entrySet`' is used in the next step, then the candidate shown in Listing 2 would be favored over the one in Listing 3.

Listing 1 An example query with part of code

```
/**
 * hashmap key iteration.
 */
public static void hashmapKeyIteration(){
    Map<String, String> map = new HashMap<String, String>();
    //get stuck here and issue the query "Hashmap key iteration in Java"
    ...
```

Listing 2 Candidate API sequence 1

```
for(Map.Entry<String, String> entry:map.entrySet()) {
   System.out.println("key="+entry.getKey()+", value="+entry.getValue());
}
```

Listing 3 Candidate API sequence 2

```
for(String key : map.keySet()) {
   System.out.println("key="+key+", value="+map.get(key));
}
```

From this example, one may argue that the context should be taken into account in conjunction with the query when making API recommendations. Current API recommendation methods rely on either the query or existing code fragment, but not both. For example, one state-of-the-art API recommender BIKER [16] can only recommend "java.util.Map.keySet" for the aforementioned query, which is the third

on the list of recommendations. A natural question arises: can we make the best of the two because naturally, they both contribute valuable information for recommending APIs that the developer craves? In this paper, we propose **c**ontext-aware **A**PI **r**ecommendation using **t**ensor **f**actorization (CARTF) to incorporate context information in query-based API recommendations. Unlike previous approaches that simply leverage API information itself, CARTF regards the enclosing client code (in particular, the statements for instantiating classes and statements for method invocation) as the context. In addition, it uses enclosing control flow relevant information related to the two types of statements to enrich context information. Undoubtedly, there are some technical challenges. A standard approach for query-based recommendation is to utilize a binary, query-API relation, which can be captured by a matrix. However, with the introduction of a context, a matrix would not be sufficient. Rather, we need to represent a triadic query-API-context relation, for which an (order-3) tensor is needed. Furthermore, one of the greatest difficulties in the standard query-based API recommendation lies in the sparsity of the available entries in the query-API matrix, so matrix completion methods have to be utilized. This issue deteriorates in the current setting as one more dimension (i.e., the context) has to be considered. To this end, CARTF employs non-negative tensor factorization to approximate missing values in the tensor [17].

When putting CARTF into use, CARTF encodes the current context and uses the Smith–Waterman algorithm [18] to identify the most similar context in the tensor, based on which a list of APIs is ranked. Since tensor factorization is computation-intensive, to reduce the cost, CARTF first retrieves the most similar historical queries from the code base and constructs the tensor. The intuition is that similar queries are usually from similar programming tasks, and thus are more likely to have target APIs. To bridge the lexical gap among the queries, CARTF uses the textual similarity metric introduced by Mihalcea *et al.* [19], which performs well in measuring the semantic similarity of short texts. To improve the performance, CARTF uses the measure of word semantic similarity based on word embedding technique [20] rather than those shown in [19].

To evaluate the effectiveness of CARTF, we select two state-of-the-art query-based API recommendation approaches, namely, RACK [9] and BIKER [10], as baselines to demonstrate the performance. To construct the query-APIs-context tuples, we resort to the popular Q&A website StackOverflow to extract useful information as adopted by the two baseline approaches. Particularly, we reuse the Q&A data published by the baselines, and manually collect 458 queries as the test dataset. We mimic the actual development process by simulating the scenario that a developer is progressively completing a program. More concretely, we consider 0%, 20%, 40%, 60%, and 80% of the program; each of these fragments provides growing context information of the code snippet. The experiments show that, in general, CARTF outperforms RACK and BIKER at different stages of the development process on a wide range of metrics, including the success rate, precision, recall, mean reciprocal rank (MRR), mean average precision (MAP), and normalized discounted cumulative gain (NDCG). In particular, for arguably the most interesting metric SuccessRate@1 (which measures the success rate of the top recommendation), on average, CARTF achieves a relative 82% improvement over RACK for the class-level recommendation and a relative 35.25% improvement over BIKER for the method-level recommendation. For the API recommendation example shown in Listing 1, CARTF can recommend "java.util.Map.entrySet", "java.util.Map.Entry.getValue", "java.util.Map.Entry.getKey", "java.util.Map.keySet" on the top list. Moreover, when the next statement has been written in Listing 2, CARTF recommends "java.util.Map.Entry.getValue" and "java.util.Map.Entry.getKey" in the first and second positions of the recommendation list.

The main contributions of this paper are summarized as follows:

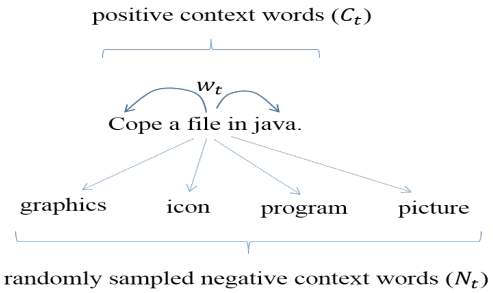


Figure 1 Positive and negative training examples in the skip-gram model

1. We propose CARTF, a novel approach that explicitly models the context information of code snippets as a tensor and harnesses it to improve query-based API recommendations. To the best of our knowledge, this is the first time that context information is explicitly modeled and incorporated into query-based API recommendations.

2. We perform an extensive, quantitative evaluation, where the experimental results confirm that CARTF can recommend APIs more accurately against a comprehensive set of metrics, considerably outperforming state-of-the-art baseline approaches.

3. We release the source code and dataset of our evaluation to help other researchers replicate and extend our study[1)].

Structure of the paper. The remainder of this paper is organized as follows: Section 2 introduces the background. Section 3 describes the technical details of our approach. Section 4 presents the experimental results. Section 5 discusses threats to validity. Section 6 presents a discussion of the related work. Section 7 concludes the paper and outlines future research plans.

2 Background

2.1 Word Embedding

Word embedding is a neural network based approach designed to transform words in a sequence into low-dimensional vectors [21], which has been successfully applied in a variety of natural language processing (NLP) tasks [22–24]. Many models have been proposed to implement word embedding, *e.g.*, continuous bag-of-words model [25], continues skip-gram model [21], etc. These models were shown to significantly outperform more traditional count-based approaches in NLP [22, 24]. It is reported that the skip-gram model is usually more accurate [21], though at the expense of a longer training time.

The skip-gram model learns vector representations of words that are useful for predicting the surrounding words in a sentence. Figure 1 illustrates the training procedure for the skip-gram model. Here, we assume a binary logistic regression model

$$\Pr(w_k \in C_t \mid w_t) = \sigma(w_t^T w_k) = \left(1 + \exp(-w_t^T w_k)\right)^{-1}$$

where w_k and w_t are the vector representations of the words. The model is trained to predict the probability of w_k being in the context C_t of w_t, by which the vector representation can then be extracted.

1) The replication package is available at https://github.com/yuierchen/CARTF.

If the word w_k is in the context, it is considered to be a positive example (w_+); any other word can serve as a negative example (w_-). The context C_t is usually defined as a fixed-size window centered at the current word w_t, whereas the set of negative examples N_t is constructed by randomly sampling from the domain vocabulary. When trained on a sequence of T words, the skip-gram model uses the stochastic gradient descent algorithm to minimize the negative of the log-likelihood objective $J(w)$ as follows:

$$J(w) = \sum_{t=1}^{T} \sum_{w_+ \in C_t} (\log \sigma(w_t^T w_+)) + \sum_{w_- \in N_t} \log \sigma(-w_t^T w_-).$$

2.2 Tensor and Decomposition

An N-th order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n}$ is of rank-1 if it can be written as the outer (aka. tensor) product of N vectors, i.e., $\mathcal{A} = a^{(1)} \circ a^{(2)} \circ \cdots \circ a^{(N)}$, where $\circ$ denotes the outer product and $a^{(n)} \in \mathbb{R}^{I_n}$ for $n = 1, \cdots, N$ is a vector. Meanwhile, each entry of the tensor $\mathcal{A}$ can be written as $\mathcal{A}_{i_1 i_2 \cdots i_N} = a_{i_1}^{(1)} \cdots a_{i_N}^{(N)}$, where $a_{i_n}^{(n)}$ is the i_n-th entry of the vector $a^{(n)}$ for $1 \leqslant n \leqslant N$. The rank of tensor $\mathcal{A}$, denoted by $R_{\mathcal{A}}$, is defined as the minimum number of rank-1 tensors required to recover $\mathcal{A}$ by summing these rank-1 tensors up.

The canonical polyadic (CP) decomposition (aka. tensor rank decomposition) decomposes a tensor into the sum of a set of rank-1 tensors. For instance, given a 3rd order tensor $\mathcal{A} \in \mathbb{R}^{I \times J \times K}$, the CP decomposition can be expressed as:

$$\mathcal{A} \approx [\![U, S, T]\!] := \sum_{r=1}^{R_{\mathcal{A}}} u_r \circ s_r \circ t_r$$

where $u_r \in \mathbb{R}^I, s_r \in \mathbb{R}^J, t_r \in \mathbb{R}^K, r = 1, 2, \ldots, R_{\mathcal{A}}$. Note that $U = [u_1, u_2, \ldots, u_{R_{\mathcal{A}}}]$, $S = [s_1, s_2, \ldots, s_{R_{\mathcal{A}}}]$ and $T = [t_1, t_2, \ldots, t_{R_{\mathcal{A}}}]$.

We further write the 3-mode of $\mathcal{A}$ as: $A_{(1)} \approx U(T \odot S)^T$, $A_{(2)} \approx S(T \odot U)^T$, $A_{(3)} \approx T(S \odot U)^T$, where $\odot$ denotes the Khatri-Rao product. The CP decomposition can be computed by, e.g., the alternative least square (ALS) algorithm:

$$\min_{U,S,T} \frac{1}{2} \|\mathcal{A} - [\![U, S, T]\!]\|_F^2,$$

where $\|\cdot\|_F$ is the Frobenius norm. The algorithm updates each factor matrix using the following equations:

$$\hat{U} = A_{(1)}(T \odot S)(T^T T * S^T S)^\dagger,$$

$$\hat{S} = A_{(2)}(T \odot U)(T^T T * U^T U)^\dagger,$$

$$\hat{T} = A_{(3)}(S \odot U)(S^T S * U^T U)^\dagger,$$

where $*$ denotes the Hadamard product of two matrices, and $\dagger$ denotes the Moore-Penrose pseudoinverse of a matrix.

To avoid overfitting, regularization terms related to U, S and T can be introduced as follows:

$$\min_{U,S,T} \frac{1}{2} \|\mathcal{A} - [\![U, S, T]\!]\|_F^2 + \frac{\lambda}{2} \left(\|U\|_F^2 + \|S\|_F^2 + \|T\|_F^2 \right),$$

where λ is the regularization parameter. The approximation of tensor $\mathcal{A}$, i.e., $\hat{\mathcal{A}}$, can be written as

$$\hat{\mathcal{A}} = \|\hat{U}, \hat{S}, \hat{T}\| = \sum_{r=1}^{R_{\mathcal{A}}} \hat{u}_r \circ \hat{s}_r \circ \hat{t}_r.$$

As the tensors considered here are non-negative, it is reasonable to add the non-negative restriction to the CP decomposition, giving rise to the non-negative CP decomposition (NNCP), i.e.,

$$\min_{\hat{U},\hat{S},\hat{T}\geqslant 0} \frac{1}{2}\|\mathcal{A}-\hat{\mathcal{A}}\|_F^2+\frac{1}{2}\left(\|\hat{U}\|_F^2+\|\hat{S}\|_F^2+\|\hat{T}\|_F^2\right),$$

where $\hat{U},\hat{S},\hat{T}\geqslant 0$ stipulates that all entries of the matrices $\hat{U},\hat{S},\hat{T}$ are non-negative.

2.3 The Smith-Waterman algorithm

The Smith-Waterman algorithm [18] for local sequence alignment is to find highly similar fragments in two sequences. Assume two sequences $A=a_1\dots a_n$ and $B=b_1\dots b_m$ where n,m are the lengths of A and B respectively. The similarity is based on two weight functions, i.e., $s(a_i,b_j)$ and w_k. The former measures the degree of "similarity" between a_i, b_j, whereas the latter represents the penalty for a vacancy of length k. These two functions are to be specified by the users according to concrete applications.

The Smith-Waterman algorithm creates a matrix $SA[n][m]$ with $SA[i,0]=SA[0,j]=0$ for $0\leqslant i\leqslant n, 0\leqslant j\leqslant m$, and proceeds as follows. For $1\leqslant i\leqslant n, 1\leqslant j\leqslant m$, let

$$SA[i][j]=\max\left\{\begin{array}{c} 0,\ SA[i-1][j-1]+s(a_i,b_j),\\ \max_{k\geqslant 1}\{SA[i-k][j]-w_k\},\\ \max_{l\geqslant 1}\{SA[i][j-l]-w_l\}\end{array}\right\}.$$

After computing the matrix $SA[n][m]$, to find the optimal alignment the algorithm starts the backtrace with the highest scoring cell in the matrix and expands by following the path through the maximum scores back until 0 is reached. In the end, the best local alignment is generated.

3 The CARTF Approach

Figure 2 illustrates the overall framework of the CARTF approach. As the first step, CARTF conducts data collecting and processing to extract useful information from StackOverflow (labeled by ① in Figure 2; cf. Section 3.1). When a user query is received, CARTF recommends APIs via three major steps:

Step I. Retrieve similar queries for the user query (labeled by ②; cf. Section 3.2);

Step II. Assemble the retrieved queries, as well as the associated contexts and APIs, to form the tensor and utilize the tensor factorization to fill in the missing values (labeled by ③; cf. Section 3.3);

Step III. Apply the Smith-Waterman algorithm to identify the most similar context in the tensor, rank the APIs accordingly based on the values of the entries in the tensor (labeled by ④; cf. Section 3.4).

The main purpose of Step I is to narrow down the scale of the candidate APIs, under the assumption that similar user queries tend to use similar APIs. In Step II, NTF is utilized to fill the empty values in the Query-API-Context tensor due to the sparsity (tensor completion). In Step III, based on the completed tensor, CARTF uses the Smith-Waterman algorithm to match the most similar context in the tensor. In the sequel, we shall articulate the details of these steps.

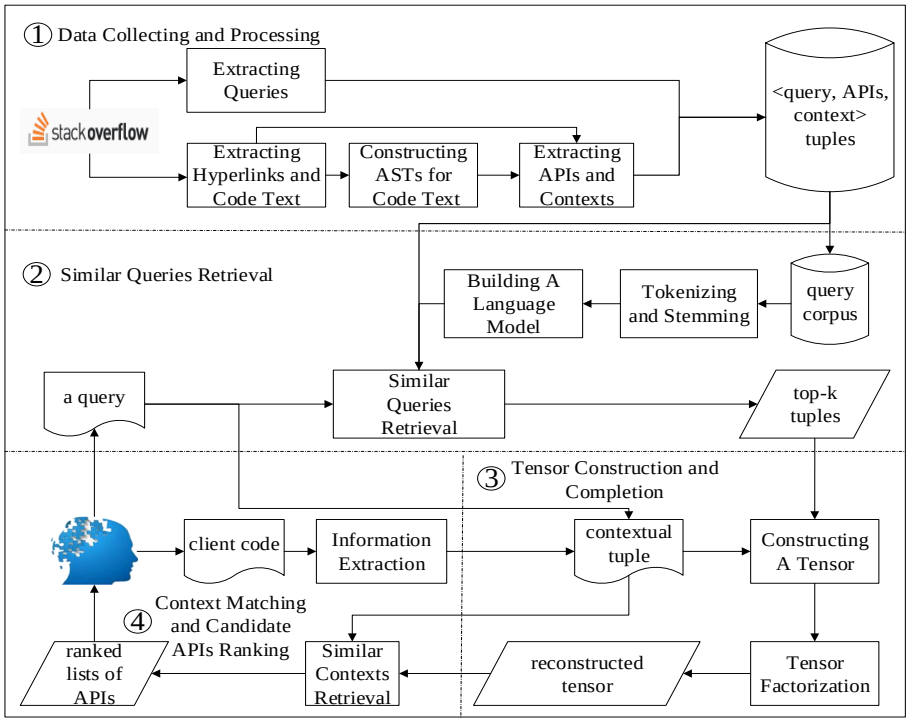


Figure 2 The overall framework of CARTF

3.1 Data Collecting and Processing

As the first step, we collect the original data from StackOverflow. We adopt the filtering method [16]. Namely, for each answer in the question, we extract a tuple $\langle query, APIs, context\rangle$ which will be the basis of the tensor construction. To ensure the high quality of data, we only keep the answers which either have positive scores or have been accepted. We now enunciate how the three components, i.e., query, APIs and context, are obtained.

We extract the question's title as the query. In addition, we extract the hyperlinks and the plain text contained in every HTML tag <code> in the answer. To concretize the notion of context, we utilize program analysis by concentrating on two types of statements, i.e., statements for instantiating a class (e.g., `new C(...)`, where `C` is a predefined class), and statements for method invocation. Intuitively, these statements usually determine, or at least strongly influence, the APIs that will be subsequently invoked. In addition, we include enclosed control flow relevant information related to these two types of statements (such as reserved identifiers `if`, `for`, `while`, `break`, etc) since they signify the execution path of the API sequences and enrich the context information.To encode the context, we design a mapping shown in Table 1. In this way, the context is represented as a string. As an example illustrated in Listing 2, the context in the method body is encoded as a string "java.util.HashMap C java.util.Map.entrySet() java.io.PrintStream.println() java.util.Map.Entry.getKey() java.util.Map.Entry.getValue() c", where 'C' represents 'ForStatement' and 'c' represents the end of the 'ForStatement'. Accordingly, we obtain the API sequence by simply removing the tags, i.e., "java.util.Map.entrySet() java.io.PrintStream.println() java.util.Map.Entry.getKey() java.util.Map.Entry.getValue()". This is done by traversing the abstract syntax trees (ASTs) built by Eclipse JDT[2)] for the plain text contained in each HTML tag <code> of the answer.

2) By Eclipse JDT Core Component, http://www.eclipse.org/jdt/core/.

Table 1 Defined mapping rules for the important AST node. Each element of AST control nodes corresponds to one character. (An uppercase character represents the statement of a significant AST node and its lowercase counterpart represents the end of the statement.)

AST Node Type	Symbol	AST Node Type	Symbol
EnhancedForStatement	A...a	ThrowStatement	H
IfStatement	B...b	SwitchStatement	I...i
ForStatement	C...c	SynchronizedStatement	J...j
ReturnStatement	D	AssertStatement	K
BreakStatement	E	CatchClause	L...l
WhileStatement	F...f	ContinueStatement	M
TryStatement	G...g	DoStatement	N...n

In order to detect the API in the answer more comprehensively, CARTF checks every hyperlink in the answer and uses regular expressions to identify the full name of the corresponding API method. For example, given the hyperlink https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#substring(int beginIndex), it extracts the API method java.lang.String.substring. We use the API extracted from the hyperlinks to expand the API extracted from the plain text contained in the HTML tag <code>.

To summarize, we collect the data and extract the $\langle query, APIs, context\rangle$ tuples from StackOverflow, which forms the basis of tensor construction in Section 3.3.

3.2 Similar Queries Retrieval

To measure the similarity of two queries, we need to build a domain-specific language model. To this end, we first tokenize the queries extracted from StackOverflow and perform stemming (i.e., transform each word to its root form[3)]). We then train a word embedding model using word2vec [21] and build the word IDF (inverse document frequency) vocabulary. IDF represents the inverse of the number of queries that contain the word, and is used as a weight on top of the word embedding. Intuitively, the more queries in which a word appears, the less likely the word carries important semantic information, so the word would carry a low IDF value.

Given two bags of words T and Q, CARTF calculates the asymmetric similarity score as

$$sim(T \longrightarrow Q) = \frac{\sum_{w\in T} sim(w, Q) * idf(w)}{\sum_{w\in T} idf(w)}$$

where $sim(w, Q)$ is the maximum value of $sim(w, w')$ for each word $w' \in Q$, and $sim(w, w')$ is the cosine similarity of the word embedding vectors of w and w'. The asymmetric similarity $sim(Q \longrightarrow T)$ is computed analogously. Intuitively, a word with lower IDF value would contribute less to the similarity score. Finally, the similarity score between T and Q is computed as the harmonic mean of the two asymmetric scores:

$$sim(T, Q) = \frac{2 * sim(T \longrightarrow Q) * sim(Q \longrightarrow T)}{sim(T \longrightarrow Q) + sim(Q \longrightarrow T)} \quad .$$

3) By the NLTK package [26].

3.3 Tensor Construction and Completion

Recall that, as per Section 3.1, the dataset is prepared as a collection of triplets $\langle query, API, context\rangle$. Given a user query, CARTF aims to represent the relevant triadic Query-API-Context relation as a tensor. To this end, CARTF first retrieves the top-k similar queries for a given user query from the dataset, where the similarity is measured according to the approach discussed in Section 3.2. The reasons of focusing on top-k similar queries are two-fold: first, for efficiency consideration, as tensor factorization is usually computation-intensive; second, for precision consideration, as too many dissimilar queries could overshallow the current query, resulting in inaccurate recommended results. In Section 4, we will empirically identify the optimal hyper-parameter k, where we set $k = 11$ for class-level API recommendation and $k = 7$ for method-level API recommendation. The obtained top-k similar queries give rise to a set $\mathcal{R}_{sim}$ of triples $\langle Query, API, Context\rangle$.

From $\mathcal{R}_{sim}$, a binary third-order tensor $\mathcal{Y} \in \mathbb{R}^{I\times J\times K}$ can be constructed, where I, J, K are the number of queries (tokenized and stemmed), APIs and contexts, respectively. Each entry of the tensor has value 1 indicating an observed assignment and 0 to indicate a missing value:

$$y_{q,a,c} := \begin{cases} 1, \text{ if}(q,a,c) \in \mathcal{R}_{sim} \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

We first load the data and create a matrix $\mathbf{Q}^{(1)}$ for the first context according to Equation (1); we then go through the context dimension to create matrices $\mathbf{Q}^{(1)}, \mathbf{Q}^{(2)}, \cdots, \mathbf{Q}^{(K)}$ for K contexts in $\mathcal{R}_{sim}$. Afterwards, we construct the tensor $\mathcal{Y} \in \mathbb{R}^{I\times J\times K}$, the slices of which are the matrices $\mathbf{Q}^{(1)}, \mathbf{Q}^{(2)}, \cdots, \mathbf{Q}^{(K)}$.

After constructing the tensor, we use the NNCP Algorithm (cf. Section 2.2)[4] to obtain the latent factor matrices $\hat{\mathbf{Q}}$, $\hat{\mathbf{A}}$, $\hat{\mathbf{C}}$, based on which the prediction value of API j from the query i at the context k is given by

$$\hat{\mathcal{Y}}_{ijk} \approx \sum_{r=1}^{R_{\mathcal{Y}}} q_r^{(i)} a_r^{(j)} c_r^{(k)}. \tag{2}$$

Notice that a suitable $R_{\mathcal{Y}}$ is crucial: increasing the number of $R_{\mathcal{Y}}$ allows to represent more factor structures so can avoid under-fitting, but a larger $R_{\mathcal{Y}}$ risks over-fitting. Empirically we set $R_{\mathcal{Y}}$ to be the half of the minimum dimension of the Query, API, and Context.

3.4 Context Matching and Candidate API Ranking

From the current query (q) and the partially completed code snippet, we can obtain the context (c) in exactly the way as what was described in Section 3.1. With the (tokenized and stemmed) query, CARTF then obtains an API-Context matrix M where each entry $M_{a,c}$ is indexed by API a and context c from the tensor $\hat{\mathcal{Y}}$.

CARTF then utilizes the Smith-Waterman algorithm (cf. Section 2.3) to compare the current context c and the contexts in the API-Context matrix M, both of which are treated as sequences. To apply the Smith-Waterman algorithm, we adopt the constant model of vacancy weights by setting $w_k = kw_1$, i.e., the penalty for a vacancy is directly proportional to the length of the vacancy (note that w_1 is the penalty

4) https://github.com/Large-Scale-Tensor-Decomposition/tensorD.

for a single vacancy). As a result, the original Smith-Waterman algorithm can be simplified to

$$SA[i][j] = \max \left\{ \begin{array}{c} 0,\ SA[i-1][j-1] + s(a_i, b_j), \\ SA[i-1][j] - w_1,\ SA[i][j-1] - w_1 \end{array} \right\}.$$

The Smith-Waterman algorithm returns the maximum matching subsequence $match(C, D)$ of two context sequences C and D, based on which the similarity score between C and D can be computed as

$$sim(C, D) = \frac{2 \cdot |match(C, D)|}{|C| + |D|},$$

where $|\cdot|$ returns the length of the sequence.

CARTF returns a vector of APIs from the API-context matrix M for the present context c. It then ranks these APIs according to the tensor entry and recommends appropriate APIs to the users.

4 Evaluation

To investigate the effectiveness of CARTF, we perform an empirical study by simulating the behavior of a developer working at different stages of a programming task on partially completed code snippets. All the experiments were conducted on a workstation equipped with two 2.6 GHz Xeon E5-2640 v3 CPUs, running Windows 10 OS.

We download the official data dump of StackOverflow (published in Dec. 9th, 2017) as BIKER did for fair comparison. Since we focus on recommendations for Java APIs, we extract 1,347,908 questions tagged with "java". To keep the data consistent, we adopt the filtering method [16] by which we collect 125,847 questions. By the approach described in Section 3.1, we extract 62,067 tuples of the form $\langle query, APIs, context \rangle$. To evaluate the effectiveness of CARTF, we directly use the test dataset used in BIKER [16]. To reflect context-aware API recommendation, the second author and the third author independently program in the IDE to solve these questions. They write different method bodies when there are multiple solutions to a programming task. Afterwards, the two programmers discuss and further expand the method bodies for the questions. In this way, we collect 458 questions as the test dataset.

We simulate different stages of a development process to study whether CARTF is applicable in real-world settings. To this end, some parts of the program are removed to mimic the real scenarios. Particularly, we take respectively 0%, 20%, 40%, 60% and 80% of the length of each program (measured in code lines) as context. Accordingly, the APIs used in the rest of the program are collected as the ground-truth $GT(q, c)$, where q is a query and c is the context information in the client code.

Overall, we collect 458, 453, 429, 375, 293 queries for class-level recommendation and 458, 455, 445, 412, 338 queries for method-level recommendation, corresponding to the 0%, 20%, 40%, 60%, 80% of the program in length respectively. They constitute the test dataset.

Baseline approaches. We compare the performance of CARTF with two state-of-the-art baseline methods, i.e., RACK [9] and BIKER [16].

RACK constructs a keyword-API mapping database where the keywords are extracted from Stack-Overflow posts and the mapped APIs are collected from the corresponding accepted answers. Based on the database, RACK recommends a ranked list of API classes for a given query expressed in natural language. Since RACK recommends APIs at the class level, to make a fair comparison, we adapt CARTF and only keep the class names from the recommended APIs.

BIKER leverages StackOverflow posts to extract candidate APIs for a programming task, and ranks the candidate APIs by considering the query's similarity with both the StackOverflow posts and API documentation. To bridge the lexical gap between the natural language description of the programming task and the API description in documentation, BIKER exploits word embedding technique to calculate the similarity scores.

4.1 Evaluation Metrics

We use $REC(q, c)$ to denote the recommended list of APIs for query q and context information c in the client code. Recall that $GT(q, c)$ denotes the ground truth. To measure the performance of API recommender systems, we consider five metrics, namely, success rate, accuracy, NDCG, MAP and MRR. In particular, MRR and MAP are standard evaluation metrics in information retrieval [27], and success rate, accuracy as well as NDCG are often used to evaluate recommendation [28, 29]. Given a ranked list of recommendations, a developer is typically interested in the top-N items only. Hence, in our evaluation, success rate, accuracy (including precision and recall), MAP, MRR and NDCG are computed by some pre-selected N. Typically, N is set to be 1, 3, 5, or 10. Namely, let $REC_N(q, c)$ be the set of top-N recommended items, and $match_N(q, c) = GT(q, c) \cap REC_N(q, c)$ be the set of items in the top-N list that match those in the ground-truth data.

Success rate. Given a set R consisting of pairs of the form (q, c), this metric measures the rate at which a recommendation engine returns at least one matched item among top-N recommended ones.

$$SuccessRate@N = \frac{\#_{(q,c)\in R}(|match_N(q, c)| > 0)}{|R|} \times 100\%$$

where $\#(\varphi)$ returns the number of times that φ evaluates true and $|R|$ is the cardinality of R.

Accuracy. We mainly use standard *precision* and *recall* to measure accuracy [28]. $Precision@N$ calculates the proportion of the top-N recommended items in the ground-truth data set, viz.

$$Precision@N = \frac{|match_N(q, c)|}{N}$$

and $Recall@N$ calculates the proportion of the ground-truth items found in the *top-N* items, viz.

$$Recall@N = \frac{|match_N(q, c)|}{|GT(q, c)|}$$

NDCG. Normalized Discounted Comulative Gain (NDCG) measures the quality of ranking by calculating the gain of each result according to its position [29]. NDCG can be calculated as follows.

$$NDCG@N = \frac{DCG@N}{idealDCG@N} \quad DCG@N = \sum_{i=1}^{N} \frac{2^{rel(i)} - 1}{log_2(i + 1)}$$

where i is the rank; $rel(i)$ is a binary function to check whether the API at rank i is correct. For example, if the API at rank i is correct, $rel(i) = 1$; otherwise, $rel(i) = 0$.

MAP. Mean Average Precision (MAP) measures the quality of rank when a query may have multiple correct answers [2, 30]. MAP is defined as the mean of the average precision values of all queries, and can be calculated as follows.

$$MAP@N = \frac{1}{|R|} \sum_{(q,c)\in R} \frac{\sum_{i=1}^{N}(P(i) \times rel(i))}{|match_N(q, c)|}$$

where $P(i) = \frac{\#\text{correct answers in top } i}{i}$, i.e., the precision at a given cut-off rank i.

MRR. Mean Reciprocal Rank (MRR) is another widely used evaluation metric to measure the quality of the rank [2, 30]. MRR is the average of the reciprocal ranks for all the queries. The reciprocal rank of a single query is the multiplicative inverse of the first correct answer. Hence, MRR can be calculated as follows.

$$MRR@N = \frac{1}{|R|} \sum_{(q,c)\in R} \frac{1}{N_{Rank(q,c)}}$$

where $N_{Rank(q,c)}$ denotes the rank position of the first correct answer in the *top-N* recommended list for (q, c).

4.2 Results

In our experiments, we primarily investigate the following three research questions (RQs).

RQ1. How effective is CARTF, i.e., how much improvement can it achieve over the baseline methods?

RQ2. How does the number of retrieved queries affect CARTF's performance?

RQ3. How efficient is CARTF for practical use?

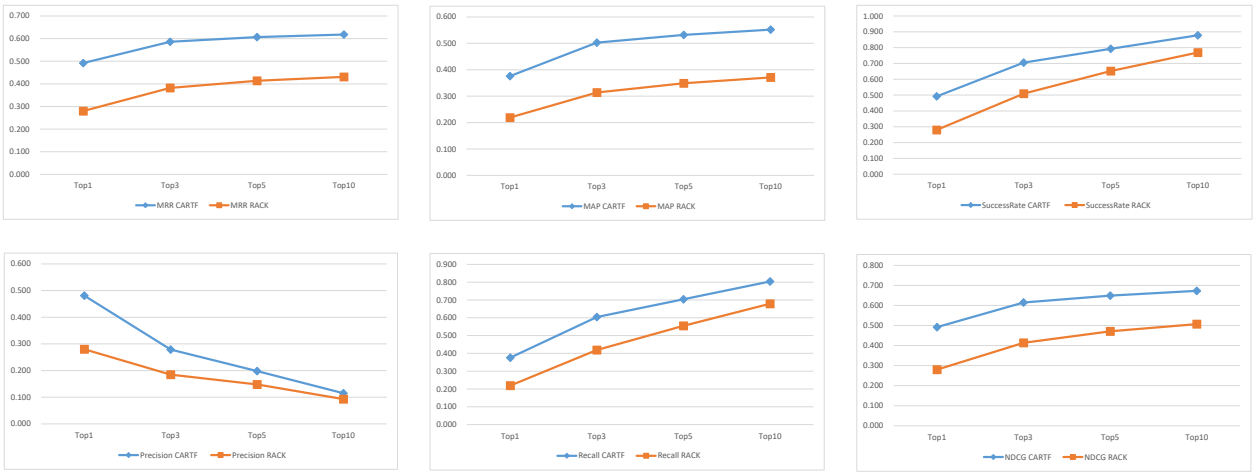


Figure 3 The performance of CARTF and RACK on Top1, Top3, Top5 and Top10 in the average metrics of MRR, MAP, Success rate, Precision, Recall, NDCG at different stages of development process. (Cf. Table 3 and Table 4 for the raw data.)

RQ1. To answer this research question, we use our test dataset to evaluate whether CARTF can outperform RACK at class-level and BIKER at method-level with respect to different stages of development process.

Figure 3 compares CARTF and RACK against various metric measures on Top1, Top3, Top5 and Top10 recommendations. We take the average measure of different stages of the development process, and original statistical results of each stage of the development process are shown in Table 3 and Table 4. One can easily observe that CARTF achieves substantially better results than RACK. The improvement is usually at the range of 10% to 40%, but is over 45% for SuccessRate@1 and MRR@N and MAP@N for all $N = 1, 3, 5, 10$. In particular, on Top 1 and Top 3 the improvements are more substantial, indicating that CARTF can put more relevant APIs on the top of the recommendation list.

Figure 4 compares CARTF and BIKER against various metric measures on Top1, Top3, Top5 and Top10 recommendations. One can observe that CARTF consistently outperforms BIKER. Table 5 and

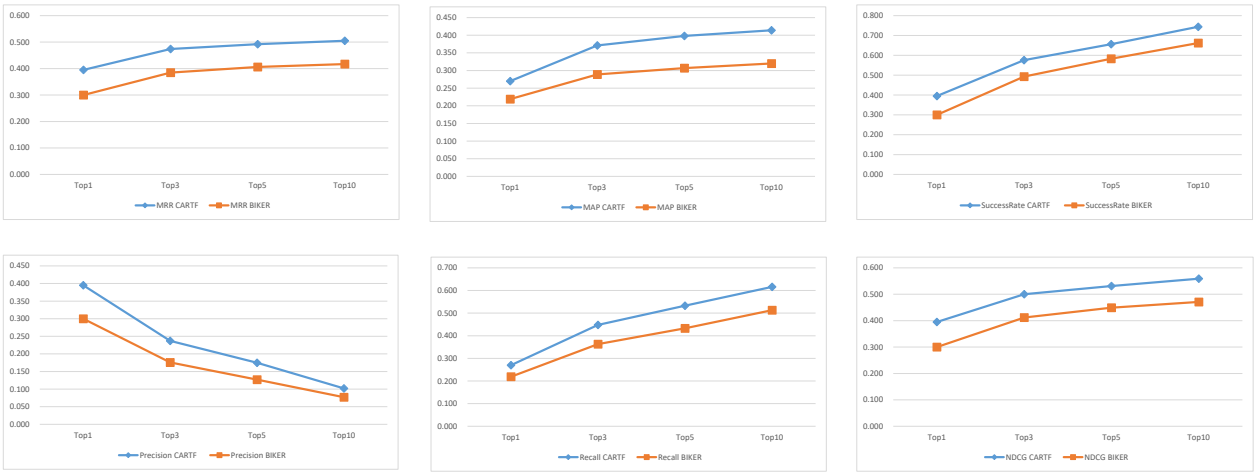


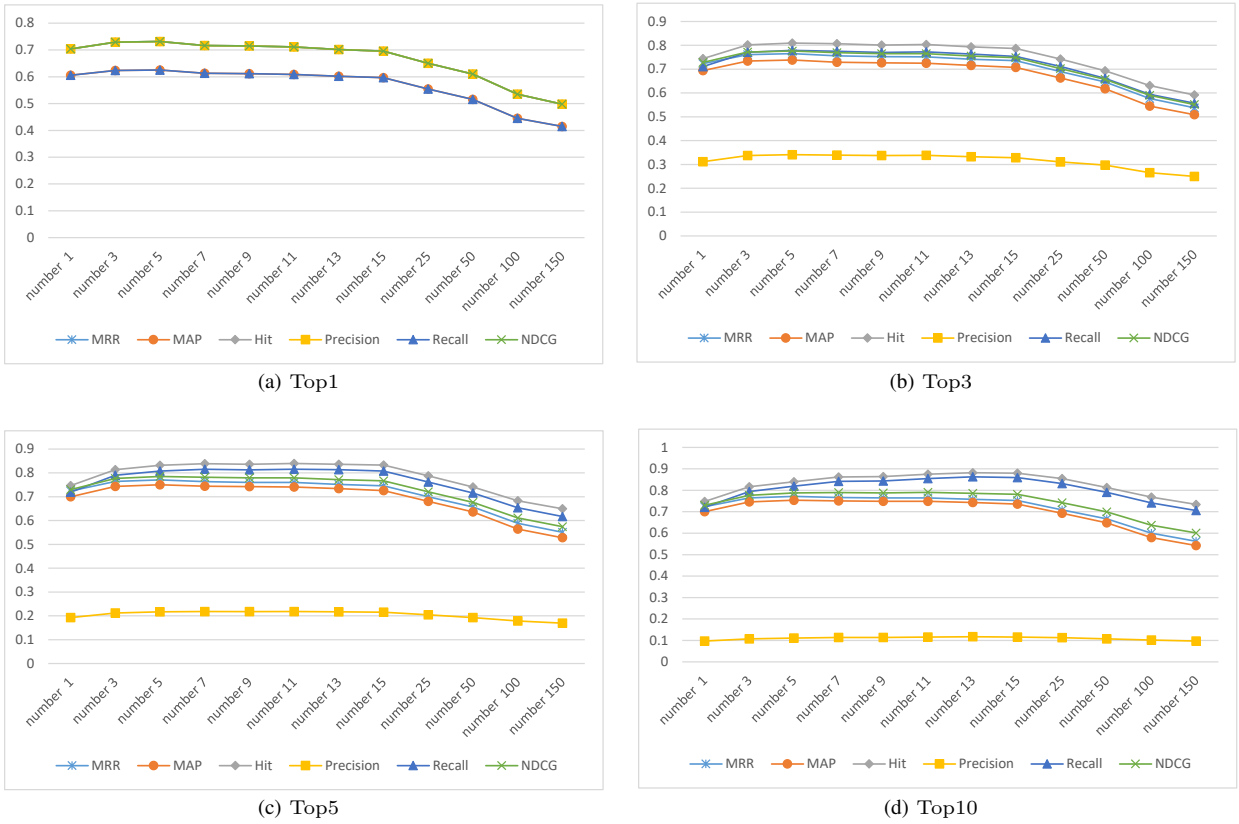
Figure 4 The performance of CARTF and BIKER on Top1, Top3, Top5 and Top10 in the average metrics of MRR, MAP, Success rate, Precision, Recall, NDCG at different stages of development process. (Cf. Table 5 and Table 6 for the raw data.)

Table 6 show the original statistical results of each stage of the development process. It is noteworthy that CARTF achieves average SuccessRate@1 of 39.5% comparing with 30.0% of BIKER. Even in the setting of 0% of the length of the context, CARTF achieves SuccessRate@1 of 47.3%, while BIKER 38.7%, which indicates that our approach outperforms BIKER even when no context information is given. BIKER considers the query's similarity with both the SO posts and the candidate API's official description. The discrepancy between the query and the API description is usually quite large, which may degrade the performance of BIKER. Instead, CARTF considers more relevant SO posts, and thus can mitigate the noise.

To sum up, CARTF can perform well in both recommending class-level and method-level APIs.

RQ2. To answer this research question, we take a search-based approach by varying #number (i.e., the number of retrieved queries) from 1 up to 150. Figure 5 and Figure 6 present the results where one can observe a consistent trend across all the metrics. Overall, the effect of API recommendation increases first and then decreases with the number of the retrieved queries. As the number of retrieved queries increases, potentially irrelevant questions may be used to construct tensors. The noise may lead to a decline in performance. As a result, the general trend is that recall increases first and then decreases with the number of retrieved questions. The optimal number for the class-level recommendation appears in the range of 11–13 and the optimal number for the method-level recommendation appears in the range of 7-9.

RQ3. To answer this research question, we record the time of CARTF and baselines in API recommendation on the testing dataset. Note that word vectors are pre-trained off-line which is largely a one-off process, so it is not the main focus of our evaluation. We are primarily concerned with recommendation time cost. Table 2 presents the time of CARTF and baselines. For the average recommendation time, CARTF achieves 67.94% and 8.44% less time than RACK and BIKER respectively. Compared with the baselines, CARTF only uses SO posts, which reduces the dimensions of tensors and thus performs faster. This means that CARTF can recommend more accurate results in a shorter time and thus is expected to be favored by developers in practical scenarios.



(a) Top1 (b) Top3

(c) Top5 (d) Top10

Figure 5 The performance of different numbers of retrieved queries on Top1, Top3, Top5 and Top10 in the average metrics of MRR, MAP, SuccessRate, Precision, Recall, NDCG at different stages of development process in class-level API recommendation

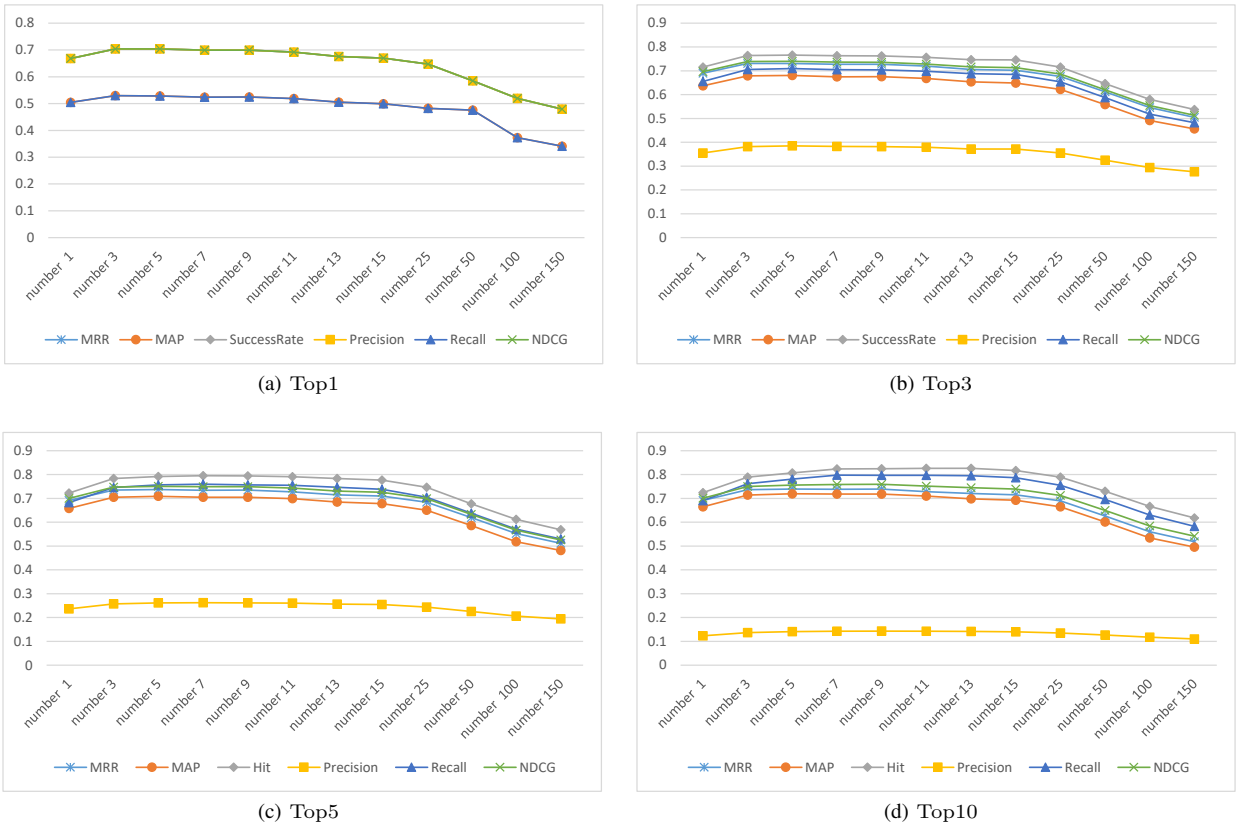
Table 2 Time cost comparison of CARTF and baselines

Class Level Recommendation	CARTF	RACK	Overhead
Time	5.92s	18.47s	-67.94%
Method Level Recommendation	CARTF	BIKER	Overhead
Time	5.53s	6.04s	-8.44%

5 Threats to Validity

Threats to internal validity are related to internal factors that could have influenced the results. The main threat is related to the errors introduced during implementation. To minimize the threats of this aspect, we double-checked and peer-reviewed our own code and reuse the implementation of the baseline tools for a fair comparison.

Threats to external validity are concerned with whether the results can be generalized to the datasets other than what were used in the experiments [31]. All APIs investigated in this paper are Java SE APIs which may not represent APIs for other libraries and programming languages. However, we argue that this is mostly an implementation limitation rather than a methodological threat. Our approach is easy to be applied to the recommendation of other Java libraries when extracting the Java APIs of the respective libraries. It would not be difficult to adapt CARTF to other programming languages, since many queries invloving other programming languages exist and could be extracted from StackOverflow.



(a) Top1 (b) Top3

(c) Top5 (d) Top10

Figure 6 The performance of different numbers of retrieved queries on Top1, Top3, Top5 and Top10 in the average metrics of MRR, MAP, SuccessRate, Precision, Recall, NDCG at different stages of development process in method-level API recommendation

6 Related Work

6.1 Textual Similarity in Software Engineering

Text retrieval techniques have been applied in various SE tasks [32]. However, system performance is usually suboptimal due to the lexical gap between the natural language and code [33]. To bridge this gap, several approaches have been recently proposed. Particularly for API recommendation, some approaches [33–38] extract API entities from the code and use the corresponding API documentation to enhance ranking results. Others [10, 16, 39–47] exploit Q&A (e.g., StackOverflow) posts to suggest APIs or code snippets.

Specifically, McMillan *et al.* [33] measured the lexical similarity between a user query and API entities and then ranked higher the code that uses API entities with higher similarity scores. Bajracharya *et al.* [34] augmented a code snippet with tokens from other code segments that use the same API entities. Ye *et al.* [38] concatenated the descriptions of all API entities used in the code and directly measured the lexical similarity between the query and concatenated document. Rahamn *et al.* proposed RACK, which constructs a keyword–API mapping database where keywords are extracted from StackOverflow questions and mapped APIs are collected from corresponding accepted answers. Based on this database, RACK recommends a ranked list of API classes for a given natural language query. BIKER [16] exploits StackOverflow posts to bridge the task–API knowledge gap and incorporates information from

StackOverflow questions and API documentation to measure the relevance of an API to the programming task description. Zhou *et al.* [48, 49] integrated users' feedback into recommendation loops and leverage learning-to-rank and active learning techniques to boost recommendation performance.

However, the approaches mentioned above do not consider the client code, which usually contains rich information for a recommendation. Some of them need well-prepared search queries that must contain keywords similar to the API names. Moreover, as a programming task description usually needs more than one API to complete, the calculation of the similarity between the programming task and one API document is not reliable, risking the overemphasis of the importance of API documentation. In CARTF, we propose to calculate the similarity between the descriptions of queries and incorporate context information into query-based API recommendations.

6.2 Recommending API Usage Patterns

Acharya *et al.* [50] presented a framework to extract API patterns as partial orders from client code. To this aim, control flow-sensitive static API traces are extracted from source code, and sequential patterns are computed. However, although this approach proposes a representation for API patterns, suggestions regarding API usage are still missing.

MAPO (mining API usage patterns from open source repositories) is a tool that mines API usage patterns from client projects [51]. The system analyzes source files to obtain API usage information and groups API methods into clusters. It then mines API usage patterns from clusters, which are ranked according to their similarity to the current development context, and recommends code snippets. Similarly, UP-Miner [52] mines API usage patterns by relying on SeqSim, a clustering strategy that reduces patterns redundancy and improves coverage. UP-Miner employs the BIDE algorithm [53] to mine API frequent closed call sequences.

Strathcona [54] mainly utilizes the structural context of existing code to retrieve similar code snippets in the repository and recommends them to developers. Different from our approach, it does not require the input of user queries. Moreover, its main purpose is to recommend similar code examples to the code under development.

Fowkes *et al.* introduced Probabilistic API Miner (PAM), a parameter-free probabilistic approach to mine API usage patterns [55]. PAM uses the structural expectation-maximization (EM) algorithm to infer the most probable API patterns from code. Niu *et al.* extracted API usage patterns using an API class or method names as queries [56]. They rely on the concept of object usage (method invocations on a given API class) to extract patterns.

NCBUP-miner (non client-based usage patterns) [57] is a technique that identifies unordered API usage patterns from the API source code, based on structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [58], which is based on vector representation and clustering, and considers the client code.

DeepAPI [59] is a deep-learning based method to generate API usage sequences given a query in the natural language. The learning problem is cast as a machine translation problem, where queries are considered the source language and API sequences as the target language. GAPI [60] uses graph neural networks to capture the high order collaborative signals from API invocations. Moreover, the work adopts context information, such as integrating structures of projects into graphs and incorporating text attributes in networks. However, the work does not consider user queries and makes recommendations solely based on the code information.

Focus [15] mines open-source project repositories to recommend API invocations and usage patterns using collaborative filtering techniques to analyze how APIs are used in projects that are similar to the current one. RecRank [61] applies a ranking-based discriminative approach leveraging API usage path features to improve the top-1 API recommendation.

Compared to these approaches, CARTF uses word-embedding techniques to retrieve similar queries and narrow down the search space of candidate APIs and considers the client code by constructing a tensor representing query-API-context triadic relations to rank and recommend APIs, which can cater for the needs of the developers better.

7 Conclusions

In this paper, we propose CARTF, a novel approach to incorporate context information into query-based API recommendations. One of our major contributions is to provide a feasible way to utilize context information to make recommendations more precise and cater better to the needs of the programmer. Our experiments have confirmed, empirically, that CARTF can substantially improve state-of-the-art query-based API recommendation approaches—at class and method levels—with an acceptable overhead, showcasing the usefulness of context information and the effectiveness of our approach.

For future work, we shall consider other forms of context information and investigate whether they could (and in the affirmative case, how to) improve the API recommendation. On the practical side, we will provide full-fledged tool support (*e.g.*, a plugin in IDE) to facilitate developers using CARTF for their programming.

Acknowledgements This work was partially supported by National Natural Science Foundation of China (NSFC, No. 61972197, No. 61802179), Collaborative Innovation Center of Novel Software Technology and Industrialization, and Qing Lan Project. T. Chen is partially supported by Birkbeck BEI School Project (EFFECT), NSFC grant (No. 61872340), and Guangdong Science and Technology Department grant (No. 2018B010107004), Natural Science Foundation of Guangdong Province, China (No. 2019A1515011689).

References

1 M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis," in *ICSE*. ACM, 2016, pp. 357–367.
2 X. Ye, H. Shen, X. Ma, R. C. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *ICSE*. ACM, 2016, pp. 404–415.
3 M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010.
4 F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on API understanding and extended boolean model (E)," in *ASE*. IEEE Computer Society, 2015, pp. 260–270.
5 F. Thung, S. Wang, D. Lo, and J. L. Lawall, "Automatic recommendation of API methods from feature requests," in *ASE*. IEEE, 2013, pp. 290–300.
6 W. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *FSE*, W. Tracz, M. P. Robillard, and T. Bultan, Eds. ACM, 2012, p. 10.
7 F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation," in *WCRE*. IEEE Computer Society, 2013, pp. 182–191.
8 C. Mcmillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *ICSE*, 2011, pp. 111–120.
9 M. M. Rahman, C. K. Roy, and D. Lo, "RACK: automatic API recommendation using crowdsourced knowledge," *CoRR*, vol. abs/1807.02953, 2018.
10 L. Cai, H. Wang, Q. Huang, X. Xia, Z. Xing, and D. Lo, "BIKER: a tool for bi-information source based API method recommendation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, 2019, pp. 1075–1079.

11 R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 117–125.

12 M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 285–294.

13 L. Ai, Z. Huang, W. Li, Y. Zhou, and Y. Yu, "Sensory: Leveraging code statement sequence information for code snippets recommendation," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 27–36.

14 A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2015.

15 P. T. Nguyen, J. D. Rocco, D. D. Ruscio, L. Ochoa, T. Degueule, and M. D. Penta, "FOCUS: a recommender system for mining API function calls and usage patterns," in *ICSE*, 2019, pp. 1050–1060.

16 Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-api knowledge gap," in *ASE*, 2018, pp. 293–304.

17 E. Frolov and I. Oseledets, "Tensor methods and recommender systems," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 7, no. 3, 2017.

18 L. Ligowski and W. Rudnicki, "An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–8.

19 R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *National Conference on Artificial Intelligence & the Eighteenth Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 2006, pp. 775–780.

20 T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013.*, 2013, pp. 3111–3119.

21 T. Mikolov, I. Sutskever, C. Kai, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

22 M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014*, 2014, pp. 238–247.

23 R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, 2011.

24 T. Mikolov, W. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics*, 2013, pp. 746–751.

25 T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR*, 2013.

26 S. Bird, "NLTK: the natural language toolkit," in *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*. The Association for Computer Linguistics, 2006.

27 N. L. An, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *IEEE/ACM International Conference on Automated Software Engineering*, 2015.

28 T. D. Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker, "Linked open data to support content-based recommender systems," in *International Conference on Semantic Systems*, 2012.

29 I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy, "Dimensions and metrics for evaluating recommendation systems," in *Recommendation Systems in Software Engineering*, 2014, pp. 245–273.

30 J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *International Conference on Software Engineering*, 2012, pp. 14–24.

31 R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey," in *SEKE*, 2010, pp. 374–379.

32 S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *35th International Conference on Software Engineering, ICSE*. IEEE Computer Society, 2013, pp. 842–851.

33 C. Mcmillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

34 S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, 2010, pp. 157–166.

35 S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, vol. 5503, 2009, pp. 385–400.

36 T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, "Enhancing software traceability by automatically expanding corpora with relevant documentation," in *IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2013, pp. 320–329.

37 J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006)*. IEEE Computer Society, 2006, pp. 195–202.

38 Y. Xin, R. Bunescu, and L. Chang, "Learning to rank relevant files for bug reports using domain knowledge," in *Acm Sigsoft International Symposium on Foundations of Software Engineering*, 2014, pp. 689–699.

39 L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. D. Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *IEEE/ACM International Conference on Software Engineering*, 2017, pp. 94–105.

40 J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Third International Workshop on Recommendation Systems for Software Engineering*, 2012, pp. 85–89.

41 L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *European Conference on Software Maintenance & Reengineering*, 2013, pp. 57–66.

42 L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Working Conference on Mining Software Repositories*, 2014, pp. 102–111.

43 M. M. Rahman, S. Yeasmin, and C. K. Roy, "Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions," in *Software Maintenance, Reengineering & Reverse Engineering*, 2014, pp. 194–203.

44 P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *International Conference on Software Engineering*, 2013, pp. 832–841.

45 W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *International Workshop on Search-driven Development: Users*, 2011.

46 C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *IEEE/ACM International Conference on Software Engineering*, 2017, pp. 392–403.

47 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.

48 Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma, and H. C. Gall, "Boosting API recommendation with implicit feedback," *IEEE Transactions on Software Engineering*, 2021.

49 Y. Zhou, H. Jin, X. Yang, T. Chen, K. Narasimhan, and H. C. Gall, "Braid: an api recommender supporting implicit user feedback," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1510–1514.

50 M. Acharya, X. Tao, P. Jian, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Joint Meeting of the European Software Engineering Conference & the Acm Sigsoft Symposium on the Foundations of Software Engineering*, 2007, pp. 25–34.

51 H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," *Proc Ecoop*, vol. 5653, pp. 318–343, 2009.

52 J. Wang, Y. Dang, H. Zhang, C. Kai, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Mining Software Repositories*, 2013, pp. 319–328.

53 J. Y. Wang and J. W. Han, "Bide: Efficient mining of frequent closed sequences," in *International Conference on Data Engineering*, 2004, pp. 79–90.

54 R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, 2006.

55 J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," *Computer Science*, pp. 254–265, 2015.

56 H. Niu, I. Keivanloo, and Y. Zou, "API usage pattern recommendation for software development," *J. Syst. Softw.*, vol. 129, pp. 127–139, 2017.

57 M. A. Saied, H. Abdeen, O. Benomar, and H. S. Diro, "Could we infer unordered api usage patterns only using the library source code?" in *IEEE International Conference on Program Comprehension*, 2015, pp. 71–81.

58 M. A. Saied, O. Benomar, H. Abdeen, and H. S. Diro, "Mining multi-level api usage patterns," in *IEEE International Conference on Software Analysis*, 2015, pp. 23–32.

59 X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., 2016, pp. 631–642.

60 C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for api usage recommendation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 36–47.

61 X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *ASE*, 2018, pp. 282–292.

8 Appendix

Table 3 MRR, MAP, SuccessRate, Precision, Recall, NDCG for class-level API recommendation

Context	Metric	Top-1			Top-3		
		CARTF	RACK	Improvement	CARTF	RACK	Improvement
0%	MRR	0.589	0.366	60.93%	0.678	0.484	40.08%
	MAP	0.379	0.246	54.07%	0.530	0.355	49.30%
	SuccessRate	0.589	0.366	60.93%	0.794	0.628	26.43%
	Precision	0.589	0.366	60.93%	0.345	0.242	42.56%
	Recall	0.379	0.246	54.07%	0.624	0.458	36.24%
	NDCG	0.589	0.366	60.93%	0.706	0.519	36.03%
20%	MRR	0.549	0.353	55.52%	0.657	0.465	41.29%
	MAP	0.362	0.246	47.15%	0.531	0.351	51.28%
	SuccessRate	0.549	0.353	55.52%	0.794	0.604	31.46%
	Precision	0.549	0.353	55.52%	0.342	0.228	50.00%
	Recall	0.362	0.246	47.15%	0.640	0.454	40.97%
	NDCG	0.549	0.353	55.52%	0.691	0.499	38.48%
40%	MRR	0.496	0.293	69.28%	0.588	0.395	48.86%
	MAP	0.379	0.236	60.59%	0.500	0.328	52.44%
	SuccessRate	0.496	0.293	69.28%	0.701	0.526	33.27%
	Precision	0.496	0.293	69.28%	0.275	0.188	46.28%
	Recall	0.379	0.236	60.59%	0.592	0.437	35.47%
	NDCG	0.496	0.293	69.28%	0.615	0.428	43.69%
60%	MRR	0.450	0.210	114.29%	0.539	0.304	77.30%
	MAP	0.391	0.190	105.79%	0.493	0.278	77.34%
	SuccessRate	0.450	0.210	114.29%	0.648	0.424	52.83%
	Precision	0.391	0.210	86.19%	0.238	0.146	63.01%
	Recall	0.391	0.190	105.79%	0.590	0.385	53.25%
	NDCG	0.450	0.210	114.29%	0.565	0.335	68.66%
80%	MRR	0.378	0.180	110.00%	0.470	0.261	80.08%
	MAP	0.368	0.177	107.91%	0.461	0.257	79.38%
	SuccessRate	0.378	0.180	110.00%	0.587	0.365	60.82%
	Precision	0.378	0.180	110.00%	0.197	0.121	62.81%
	Recall	0.368	0.177	107.91%	0.575	0.360	59.72%
	NDCG	0.378	0.180	110.00%	0.500	0.288	73.61%
Average	MRR	0.492	0.280	82.00%	0.586	0.382	57.52%
	MAP	0.376	0.219	75.10%	0.503	0.314	61.95%
	SuccessRate	0.492	0.280	82.00%	0.705	0.509	40.96%
	Precision	0.481	0.280	76.39%	0.279	0.185	52.93%
	Recall	0.376	0.219	75.10%	0.604	0.419	45.13%
	NDCG	0.492	0.280	82.00%	0.615	0.414	52.09%

Table 4 MRR, MAP, SuccessRate, Precision, Recall, NDCG for class-level API recommendation

Context	Metric	Top-5			Top-10		
		CARTF	RACK	Improvement	CARTF	RACK	Improvement
0%	MRR	0.695	0.515	34.95%	0.706	0.528	33.71%
	MAP	0.563	0.392	43.62%	0.589	0.417	41.25%
	SuccessRate	0.871	0.762	14.30%	0.947	0.849	11.54%
	Precision	0.245	0.189	29.63%	0.146	0.118	23.73%
	Recall	0.719	0.586	22.70%	0.824	0.697	18.22%
	NDCG	0.729	0.571	27.67%	0.747	0.595	25.55%
20%	MRR	0.671	0.495	35.56%	0.682	0.508	34.25%
	MAP	0.560	0.385	45.45%	0.582	0.409	42.30%
	SuccessRate	0.854	0.735	16.19%	0.938	0.827	13.42%
	Precision	0.237	0.176	34.66%	0.139	0.110	26.36%
	Recall	0.720	0.578	24.57%	0.825	0.689	19.74%
	NDCG	0.713	0.549	29.87%	0.733	0.575	27.48%
40%	MRR	0.611	0.425	43.76%	0.622	0.441	41.04%
	MAP	0.529	0.361	46.54%	0.551	0.382	44.24%
	SuccessRate	0.801	0.659	21.55%	0.885	0.773	14.49%
	Precision	0.196	0.147	33.33%	0.115	0.092	25.00%
	Recall	0.695	0.564	23.23%	0.804	0.680	18.24%
	NDCG	0.655	0.482	35.89%	0.678	0.517	31.14%
60%	MRR	0.559	0.337	65.88%	0.570	0.358	59.22%
	MAP	0.520	0.312	66.67%	0.534	0.332	60.84%
	SuccessRate	0.738	0.570	29.47%	0.821	0.717	14.50%
	Precision	0.169	0.121	39.67%	0.095	0.077	23.70%
	Recall	0.694	0.528	31.44%	0.780	0.664	17.47%
	NDCG	0.601	0.396	51.77%	0.627	0.444	41.22%
80%	MRR	0.497	0.300	65.67%	0.510	0.320	59.38%
	MAP	0.488	0.294	65.99%	0.503	0.315	59.68%
	SuccessRate	0.703	0.532	32.14%	0.798	0.679	17.53%
	Precision	0.142	0.106	33.96%	0.081	0.068	19.12%
	Recall	0.691	0.520	32.88%	0.788	0.667	18.14%
	NDCG	0.547	0.357	53.22%	0.579	0.406	42.61%
Average	MRR	0.607	0.414	49.16%	0.618	0.431	45.52%
	MAP	0.532	0.349	53.65%	0.552	0.371	49.66%
	SuccessRate	0.793	0.652	22.73%	0.878	0.769	14.30%
	Precision	0.198	0.148	34.25%	0.115	0.093	23.58%
	Recall	0.704	0.555	26.96%	0.804	0.679	18.36%
	NDCG	0.649	0.471	39.68%	0.673	0.507	33.60%

Table 5 MRR, MAP, SuccessRate, Precision, Recall, NDCG for method-level API recommendation

Context	Metric	Top-1			Top-3		
		CARTF	BIKER	Improvement	CARTF	BIKER	Improvement
0%	MRR	0.473	0.386	22.54%	0.560	0.490	14.29%
	MAP	0.267	0.241	10.79%	0.384	0.324	18.52%
	SuccessRate	0.473	0.386	22.54%	0.672	0.620	8.39%
	Precision	0.473	0.386	22.54%	0.299	0.232	28.88%
	Recall	0.267	0.241	10.79%	0.460	0.403	14.14%
	NDCG	0.473	0.386	22.54%	0.587	0.521	12.67%
20%	MRR	0.450	0.369	21.95%	0.536	0.472	13.56%
	MAP	0.266	0.240	10.83%	0.384	0.322	19.25%
	SuccessRate	0.450	0.369	21.95%	0.643	0.602	6.81%
	Precision	0.450	0.369	21.95%	0.284	0.223	27.35%
	Recall	0.266	0.240	10.83%	0.453	0.400	13.25%
	NDCG	0.450	0.369	21.95%	0.564	0.504	11.90%
40%	MRR	0.413	0.319	29.47%	0.496	0.413	20.10%
	MAP	0.273	0.232	17.67%	0.384	0.307	25.08%
	SuccessRate	0.413	0.319	29.47%	0.604	0.532	13.53%
	Precision	0.413	0.319	29.47%	0.248	0.188	31.91%
	Recall	0.273	0.232	17.67%	0.466	0.387	20.41%
	NDCG	0.413	0.319	29.47%	0.523	0.443	18.06%
60%	MRR	0.351	0.235	49.36%	0.423	0.309	36.89%
	MAP	0.281	0.201	39.80%	0.370	0.262	41.22%
	SuccessRate	0.351	0.235	49.36%	0.519	0.400	29.75%
	Precision	0.351	0.235	49.36%	0.202	0.135	49.63%
	Recall	0.281	0.201	39.80%	0.450	0.334	34.73%
	NDCG	0.351	0.235	49.36%	0.448	0.332	34.94%
80%	MRR	0.289	0.189	52.91%	0.357	0.243	46.91%
	MAP	0.265	0.181	46.41%	0.335	0.230	45.65%
	SuccessRate	0.289	0.189	52.91%	0.440	0.310	41.94%
	Precision	0.289	0.189	52.91%	0.154	0.103	49.51%
	Recall	0.265	0.181	46.41%	0.411	0.290	41.72%
	NDCG	0.289	0.189	52.91%	0.378	0.260	45.38%
Average	MRR	0.395	0.300	35.25%	0.474	0.385	26.35%
	MAP	0.270	0.219	25.10%	0.371	0.289	29.95%
	SuccessRate	0.395	0.300	35.25%	0.576	0.493	20.08%
	Precision	0.395	0.300	35.25%	0.237	0.176	37.46%
	Recall	0.270	0.219	25.10%	0.448	0.363	24.85%
	NDCG	0.395	0.300	35.25%	0.500	0.412	24.59%

Table 6 MRR, MAP, SuccessRate, Precision, Recall, NDCG for method-level API recommendation

Context	Metric	Top-5			Top-10		
		CARTF	BIKER	Improvement	CARTF	BIKER	Improvement
0%	MRR	0.576	0.515	11.84%	0.588	0.526	11.79%
	MAP	0.418	0.345	21.16%	0.438	0.362	20.99%
	SuccessRate	0.744	0.727	2.34%	0.829	0.812	2.09%
	Precision	0.223	0.168	32.74%	0.132	0.103	28.16%
	Recall	0.546	0.479	13.99%	0.634	0.571	11.03%
	NDCG	0.613	0.565	8.50%	0.638	0.585	9.06%
20%	MRR	0.558	0.497	12.27%	0.569	0.508	12.01%
	MAP	0.421	0.341	23.46%	0.437	0.357	22.41%
	SuccessRate	0.740	0.709	4.37%	0.824	0.791	4.17%
	Precision	0.217	0.161	34.78%	0.126	0.097	29.90%
	Recall	0.559	0.476	17.44%	0.637	0.564	12.94%
	NDCG	0.601	0.548	9.67%	0.627	0.567	10.58%
40%	MRR	0.514	0.435	18.16%	0.528	0.447	18.12%
	MAP	0.409	0.325	25.85%	0.425	0.338	25.74%
	SuccessRate	0.683	0.626	9.11%	0.779	0.710	9.72%
	Precision	0.179	0.134	33.58%	0.105	0.080	31.25%
	Recall	0.545	0.458	19.00%	0.631	0.540	16.85%
	NDCG	0.554	0.481	15.18%	0.584	0.506	15.42%
60%	MRR	0.440	0.327	34.56%	0.451	0.337	33.83%
	MAP	0.392	0.278	41.01%	0.403	0.288	39.93%
	SuccessRate	0.594	0.478	24.27%	0.672	0.553	21.52%
	Precision	0.145	0.099	46.46%	0.083	0.058	43.10%
	Recall	0.525	0.399	31.58%	0.594	0.467	27.19%
	NDCG	0.478	0.364	31.32%	0.504	0.387	30.23%
80%	MRR	0.374	0.258	44.96%	0.388	0.267	45.32%
	MAP	0.352	0.245	43.67%	0.366	0.255	43.53%
	SuccessRate	0.517	0.375	37.87%	0.615	0.443	38.83%
	Precision	0.109	0.075	45.33%	0.065	0.045	44.44%
	Recall	0.488	0.354	37.85%	0.584	0.423	38.06%
	NDCG	0.410	0.287	42.86%	0.442	0.309	43.04%
Average	MRR	0.492	0.406	24.36%	0.505	0.417	24.21%
	MAP	0.398	0.307	31.03%	0.414	0.320	30.52%
	SuccessRate	0.656	0.583	15.59%	0.744	0.662	15.27%
	Precision	0.175	0.127	38.58%	0.102	0.077	35.37%
	Recall	0.533	0.433	23.97%	0.616	0.513	21.22%
	NDCG	0.531	0.449	21.50%	0.559	0.471	21.67%