Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We?

Yun Peng[†], Shuqing Li[†], Wenwei Gu[†], Yichen Li[†], Wenxuan Wang[†], Cuiyun Gao[*‡§¶], and Michael Lyu[†]
[†]The Chinese University of Hong Kong, Hong Kong, China
[‡]Harbin Institute of Technology, Shenzhen, China
[§]Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen, China
[¶]Peng Cheng Laboratory, Shenzhen, China
{ypeng, sqli21, wwgu21, ycli21, wxwang, lyu}@cse.cuhk.edu.hk, gaocuiyun@hit.edu.cn

Abstract—Application Programming Interfaces (APIs), which encapsulate the implementation of specific functions as interfaces, greatly improve the efficiency of modern software development. As the number of APIs grows up fast nowadays, developers can hardly be familiar with all the APIs and usually need to search for appropriate APIs for usage. So lots of efforts have been devoted to improving the API recommendation task. However, it has been increasingly difficult to gauge the performance of new models due to the lack of a uniform definition of the task and a standardized benchmark. For example, some studies regard the task as a code completion problem, while others recommend relative APIs given natural language queries. To reduce the challenges and better facilitate future research, in this paper, we revisit the API recommendation task and aim at benchmarking the approaches. Specifically, the paper groups the approaches into two categories according to the task definition, i.e., query-based API recommendation and code-based API recommendation. We study 11 recently-proposed approaches along with 4 widely-used IDEs. One benchmark named APIBench is then built for the two respective categories of approaches. Based on APIBench, we distill some actionable insights and challenges for API recommendation. We also achieve some implications and directions for improving the performance of recommending APIs, including appropriate query reformulation, data source selection, low resource setting, user-defined APIs, and query-based API recommendation with usage patterns.

Index Terms—API recommendation, benchmark, empirical study

1 Introduction

Application Programming Interfaces (APIs) provided by software libraries or frameworks play an important role in modern software development. Almost all programs, even the basic "hello world!" program, include at least one API. However, there are a huge number of APIs from different modules or libraries. For example, the Java standard library [51] provides more than 30,000 APIs. It is therefore infeasible for developers to be familiar with all APIs. To address this problem, many approaches are proposed to recommend APIs based on input queries, which describe the programming task in natural language, or surrounding context, i.e., the code already written by developers.

However, a uniform definition of the current API recommendation task is still absent, making the task hard to be followed by potential researchers. Some studies [7], [32], [46], [60], [62] regard the task as a code completion problem, and recommend any code tokens including APIs. These studies focus on improving the prediction results of all the tokens instead of only APIs. Some studies [23], [27], [38], [56], [58] recommend relative APIs on different levels given natural language queries. Besides, the evaluation results are difficult to be reproduced by future related work. For example, for query-based API recommendation, manual evaluation is generally adopted, so the performance reported by different studies can hardly be aligned. Comparing with widely-used Integrated Development Enviroments (IDEs) or search engines is another commonly adopted yet inconsistent evaluation strategy in previous research. Therefore, to better facilitate future exploration of the API recommendation task, in this paper, we summarize the recent related approaches and build a general benchmark named APIBench.

To facilitate the benchmark creation, we group the recent related approaches into two categories according to the task definition: query-based API recommendation and code-based API recommendation:

1) **query-based API recommendation.** Approaches for query-based API recommendation aim at providing related APIs to developers given a query that describes programming requirements in natural language. The approaches can inform developers which API to use for a programming task.

2) **code-based API recommendation.** Approaches for code-based API recommendation aim at predicting the next API given the code surrounding the point of prediction. They can directly improve the efficiency of coding.

Besides the unreproducible evaluation, the two groups of studies face their own challenges. 1) For query-based approaches, high-quality queries play a critical role in accurate recommendation. However, there may exist a knowledge gap between developers and API designers in choosing terms for describing queries or APIs. For example, developers who do not know the term *"heterogeneous list"* in API documents would use other words such as *"list with dif-*

* *corresponding author.*

ferent types of elements" in the query. Whether current query reformulation techniques are effective for API recommendation and how effective it is are still remaining unexplored. 2) For code-based approaches, the quality of code before the recommendation point also affects the recommendation performance. Generally, the approaches are evaluated by simulating an actual development, i.e., some parts of a project are removed for imitating a limited context. The APIs to recommend may be located in the front, middle, or back of the code, so exploring the impact of different recommendation points is important for understanding the recommendation capability of existing approaches. Other factors such as whether the APIs are standard or user-defined, lengths of given context, and different domains can also influence the recommendation performance, which have not yet been fully investigated.

To comprehensively understand the above challenges and align the performance of current approaches, we first build a benchmark named APIBENCH. APIBENCH is built on Python and Java, and involves two datasets for evaluation, named as APIBENCH-Q and APIBENCH-C for query-based and code-based approaches, respectively. APIBENCH-Q contains 6,563 Java queries and 4,309 Python queries obtained from Stack Overflow and API tutorial websites. APIBENCH-C contains 1,477 Java projects with 1,229,698 source files and 2,223 Python projects with 414,753 source files obtained from GitHub. Based on APIBENCH, we study the following research questions:

RQ1: How effective are current query-based and code-based API recommendation approaches?
RQ2: What is the impact of query reformulation techniques on the performance of query-based API recommendation?
RQ3: What is the impact of different data sources on the performance of query-based API recommendations?
RQ4: How well do code-based approaches recommend different kinds of APIs?
RQ5: What is the performance of code-based approaches in handling different contexts?
RQ6: How well do code-based approaches perform in cross-domain scenarios?

APIBENCH involves the implementation of the related approaches proposed in the recent five years, specifically including five query-based approaches and five code-based approaches. In RQ1, we compare the performances of the approaches in APIBENCH. To answer RQ2 and RQ3, we apply four popular query reformulation techniques to the queries of APIBENCH-Q and observe the performance of the query-based approaches given reformulated queries. To answer RQ4 to RQ6, we analyze the APIs in APIBENCH-C from different aspects and study the performance of code-based approaches under different experimental settings.

Key Findings. Through the large-scale empirical study, we achieve some findings and summarize the key findings as below.

(1) For query-based API recommendation:

While current approaches make a good progress on class-level recommendation, recommending the exact API methods is still a challenging task.

Query reformulation techniques, including query expansion and query modification, are quite effective in improving the performance of query-based approaches.

Adding data sources such as Q&A forums and tutorials that are more similar to real-world queries can significantly improve the performance of current approaches.

(2) For code-based API recommendation:

Recent deep learning models such as Transformers show superior performance on this task. Meanwhile, current IDEs can achieve competitive performance as recent pattern-based and learning-based approaches. They work far more than just recommending APIs based on alphabet orders.

Current approaches are effective to recommend APIs from standard libraries and popular third-party libraries, but their performance drops a lot when recommending user-defined or project-specific APIs.

Approaches trained on one single domain face the problem of cross-domain adaptation. Approaches trained on multiple domains achieve satisfying performance when testing on most single domains, and they even outperform those trained on corresponding single domains.

Based on the findings, we conclude some implications and suggestions that would benefit future research. On the one hand, query-based API recommendation approaches should be built along with query reformulation techniques to handle queries with different qualities. We also encourage future work to leverage different data sources and few-shot learning methods to address the low resource challenge in query-based API recommendation. On the other hand, we suggest future code-based API recommendation approaches focus on improving the performance of recommending user-defined APIs as it is currently the major bottleneck.

Contributions. To sum up, our contribution can be concluded as follows.

To the best of our knowledge, we are the first to systematically study both query-based and code-based API recommendation techniques on two large-scale datasets including Java and Python.

We build an open-sourced benchmark named APIBENCH to fairly evaluate query-based and code-based approaches.

We study how different settings can impact the performance of current approaches, including query quality, cross domain adaptation, etc.

We conclude some findings and implications that would be important for future research in API recommendation.

The rest of this paper is organized as follows. We present the background and regular API recommendation process in Section 2. We describe the details of APIBENCH, current baselines and evaluation metrics in Section 3. Then we introduce the experiment results and potential findings on query-based and code-based API recommendation in Section 4 and Section 5, respectively. Based on the findings, we conclude

some implications and future directions in Section 6. Finally, we discuss threats to validity and conclusion in Section 7 and Section 8, respectively.

2 Background and Related Work

In this section, we summarize the query-based approaches and code-based approaches, respectively.

2.1 Query-Based API Recommendation Methods

We describe the typical query-based API recommendation process in Figure 1. Given a query "*Calculate int value square root*", query reformulation techniques first modify the query as "*return int value square root*" or expand it as "*finally calculate int value square root*". A knowledge base built upon available data sources is also prepared for API candidate selection. Based on the knowledge base, retrieval-based methods or learning-based methods recommend the APIs relevant to the queries.

2.1.1 Query Reformulation Techniques

Input queries can be short in length or vague in semantics. Besides, there may exist a knowledge gap between developers and search engines in query description. For rendering search engines better understand the query semantics, query reformulation is a common pre-processing method. In general, there are two major types of query reformulation approaches:

1) query expansion, which adds extra information to the original queries;

2) query modification, which modifies, replaces or deletes some words in the original queries.

Query expansion. Query expansion aims at identifying important words that are missing in the input queries. The topic is originally stemmed from the field of natural language processing (NLP). For example, the work [39] utilizes word embeddings to map words in the vector space and finds similar words to enrich the queries. For the API recommendation task, since APIs are encapsulated and organized according to classes and modules, class names and module names are important hints for recommendation. Rahman *et al.* [56], [58] propose to use keyword-API class co-occurrence frequencies and keyword-keyword co-occurrence frequencies to build the relationship between words and API classes, and add the suggested API class for query expansion.

Query modification. Query modification aims at mitigating both the lexical gap and knowledge gap between the user queries and descriptions in knowledge base. The lexical gap, such as mis-spelling, can be easily addressed by spelling correction and synonym search, etc. Recent work focuses on how to mitigate the knowledge gap by replacing inappropriate words in queries. Mohammad *et al.* [4] extract important tokens in code, and Sirres *et al.* [65] leverage discussions and code from Stack Overflow posts to build a knowledge base. Cao *et al.* [8] collect query reformulation history from Stack Overflow and propose a Transformer-based approach to learn how developers change their queries when search engines do not return desired results.

2.1.2 Recommendation with Knowledge Base

Knowledge base. API recommendation approaches generally require a knowledge base that contains all the existing APIs as the search space. There are three primary sources for the knowledge base creation, including: 1) official documentations which contain comprehensive descriptions about the API functionality and structure. 2) Q&A forums, which provide the purposes of APIs and different API usage patterns. Many studies [27], [55] leverage the Q&A pairs from Stack Overflow to select API candidates. 3) Wiki sites, which describe concepts that link different APIs. For example, Liu *et al.* [38] utilizes API concepts from Wikipedia to help build API knowledge graphs.

Retrieval-based methods. Retrieval-based methods retrieve API candidates from the knowledge base and then rank the candidate APIs by calculating the similarities between queries and APIs. For example, Rahman *et al.* [56], [58] utilize the keyword-API occurrence frequencies and API-API occurrence frequencies to find the most relevant APIs. Huang *et al.* [27] first identify the similar posts from Stack Overflow by computing query-documentation similarities and choose the APIs mentioned in posts as candidates. Liu *et al.* [38] build an API knowledge graph to represent relationships between APIs and then calculate the similarities between queries and certain parts of API knowledge graph to rank the APIs.

Learning-based methods. Another type of method is to automatically learn the relationships between queries and APIs based on deep learning techniques. The knowledge base provides query-API pairs as the ground truth. For example, Gu *et al.* [23] formulate the task as a translation problem in which a model is built to translate word sequences into API sequences. They propose an RNN model with a encoder-decoder structure to implement the translation.

2.2 Code-Based API Recommendation Methods

We describe the workflow of code-based API recommendation in Figure 2. Given a target code, context representation is an essential step. Based on the extracted context, pattern-based methods or learning-based methods are adopted by previous studies to recommend the next API.

2.2.1 Context for the Target Code

Most code-based API recommendation methods regard the code before the recommendation point as the context. We name such context as *internal context* since it only considers code in the current source code or current function body. For example, Line 1 $\sim$ 6 of the target code in Figure 2 belongs to internal context. Xie *et al.* [70] find that replacing external APIs in code (such as `Arrays:asList()` in Figure 2) with their implementations can help the identification of common usage patterns. They propose to build a hierarchical context by integrating the implementation out of the current source file. We name the implementation of external APIs as *external context*.

2.2.2 Context Representation

We divide the context representation methods into two types, i.e., pattern-based representation and learning-based representation. **Pattern-based representations** [12], [48],

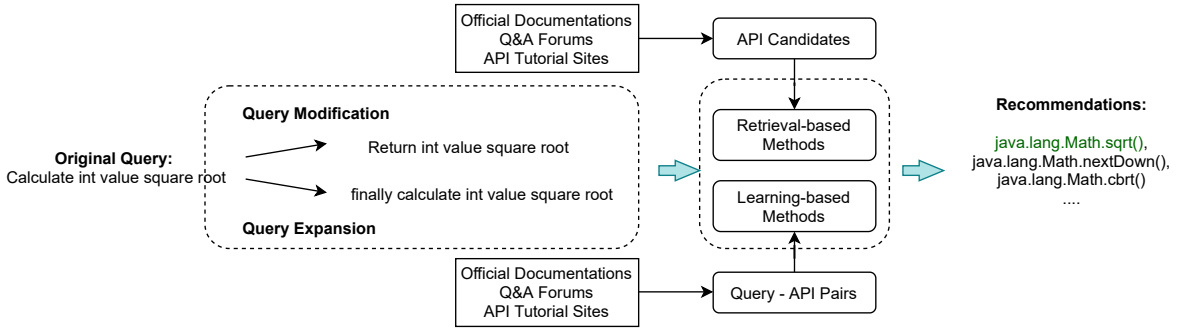


Fig. 1. The typical query-based API recommendation framework.

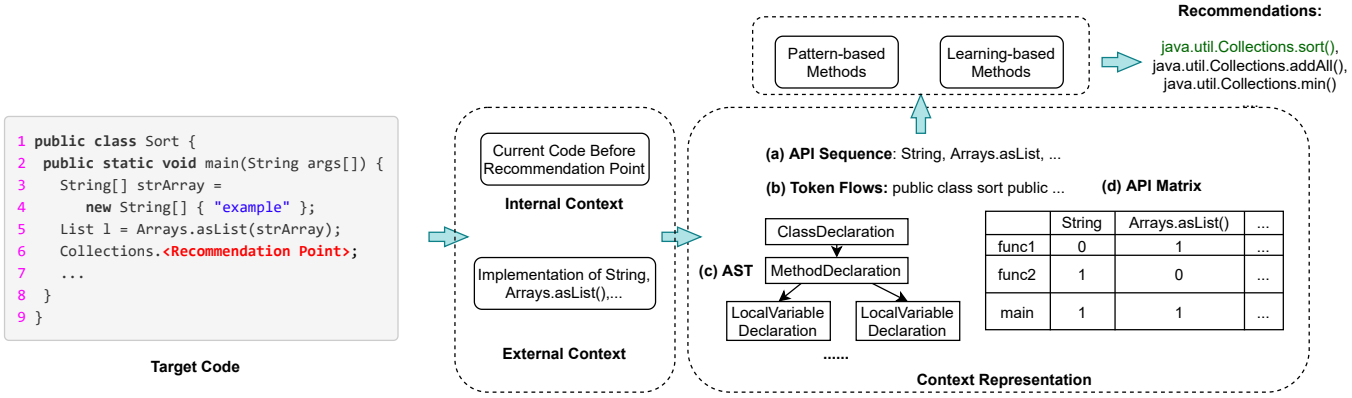


Fig. 2. The typical code-based API recommendation framework.

[49], [69], [70] do not consider all the code tokens. Instead, they only identify APIs to build API usage sequences, as shown in Figure 2 (a), API matrix, as shown in Figure 2 (d), or API dependency graphs to represent the current context. **Learning-based representations** [24], [26], [32], [60], [67] usually represent the context with token flows, as illustrated in Figure 2 (b), or other syntax structures such as Abstract Syntax Trees (ASTs), as illustrated in Figure 2 (c).

2.2.3 Recommendation Based on Context

Pattern-based methods. API recommendation is inherently a recommendation task, so some studies [12], [49] follow the collaborative filtering (*user-item*) methodology of traditional recommendation systems [61]. As shown in Figure 2 (d), they regard the internal context as the *users* and APIs as the *items*. They then calculate the similarities between different *users* to find the most similar API for recommendation. However, the methods do not consider the relationships between APIs. More recent work [69], [70] build API dependency graphs or mines association rules to capture API usage patterns.

Learning-based methods. Hindle *et al.* [26] discover the naturalness of software, rendering it possible to deploy machine learning or deep learning methods on code. Different from pattern-based methods that consider the relationships between API occurrences, learning-based methods regard API as a single code token, and reformulate the code-based API recommendation problem into a *next token prediction* problem. Many statistical language models [47], [59], [60], [67] are proposed to predict the next code token. Besides using the token sequences, more recent work [24], [32] try to leverage syntax and data flow information for more accurate prediction.

TABLE 1
Statistics of APIBENCH-Q. Ori. represent the original queries, Exp. represent the expanded queries produced by query expansion techniques, Mod. represent the modified queries produced by query modification techniques.

PL	Stack Overflow			Tutorial Websites		
	Ori.	Exp.	Mod.	Ori.	Exp.	Mod.
Python	1,925	78,157	100,100	2,384	95,360	123,968
Java	1,320	80,343	68,640	5,243	319, 783	272,636

Note that we do not aim to provide a comprehensive summary about all query-based and code-based API recommendation approaches but to choose some representatives to describe the general workflow in this section. For a more comprehensive literature review, we refer the readers to previous surveys and empirical studies [34], [63], [64].

3 METHODOLOGY

In this section, we introduce the scope of the studied APIs, the preparation of benchmark datasets, and implementation details.

TABLE 2
Statistics of Benchmark APIBENCH-C. The data includes both the training set and testing set.

PL	Domain	#Projects	#Files	LOC (per func)	#API (per func)	Total number of APIs (only testset)			LOC Threshold of Short Func	LOC Threshold of Long Func
						Standard	User-defined	Popular		
Python	General	899	230,064	15.24	5.55	1,363,240	1,747,878	54,244	8.875	54.875
	ML	323	46,556	13.89	6.08	629,437	339,821	125,377	12.65	46.05
	Security	126	15,785	18.98	6.72	111,393	64,809	3,613	6	86.5
	Web	568	82,771	14.14	5.05	369,114	241,602	11,832	7.35	51.625
	DL	307	39,577	14.58	6.25	413,295	220,228	76,654	11.675	52.525
Java	General	935	1,056,790	11.16	4.06	5,164,481	3,808,124	36,178	6.26	19.2
	Android	377	87,468	8.24	2.91	517,461	267,141	75,069	7.28	16.8
	ML	52	41,377	12.82	4.77	194,013	136,963	0	7.52	19.74
	Testing	55	23,618	9.93	3.98	105,577	55,241	22	6.44	15.68
	Security	58	20,445	12.35	5.32	125,558	74,471	1,243	6.88	20.78

3.1 Scope of APIs

To fairly compare the current API recommendation approaches, benchmark datasets should be prepared, during which the scope of studied APIs firstly needs to be defined. In this work, we focus our evaluation on two popular programming languages, i.e., Python and Java.

For facilitating the analysis of the challenges in API recommendation, we divide all APIs into *standard APIs*, *user-defined APIs*, and *popular third-party APIs*. The *standard APIs* refer to the APIs that are clearly defined and built-in in corresponding programming languages while the *user-defined APIs* are defined and used in projects along with *popular third-party APIs*. Following previous work [27], [38], [56], [58] we evaluate query-based API recommendation methods only on the *standard APIs* since currently *standard APIs* have the most comprehensive documentations and extensive discussions to build the knowledge base. We evaluate code-based API recommendation methods on all three kinds of APIs. The details of each kind for different programming languages are depicted below.

(1) Standard Java APIs. We choose the version Java 8 for our analysis since it is the most widely-used version in current projects according to the 2020 JVM Ecosystem Report [66]. We collect 34,072 APIs from the Java documentation [51] as *standard APIs*.

(2) Android APIs. We choose APIs from the Android library [20] since Android is the most popular application of Java programs. We collect 11,802 APIs from the official documentation of Android in total.

(3) Standard Python APIs. As Python Software Foundation has stopped the support for Python 2, currently only 6% of developers are still using Python 2, according to the development survey conducted by Jetbrains [28]. Considering that APIs of different versions above 3.0 are similar, we choose the newest version 3.9 to ensure the compatibility, and collect 5,241 APIs from Python standard library [53] as the *standard APIs*.

(4) Popular Python third-party APIs. Python is well extended by a lot of third-party modules. We choose five widely-used modules with sufficient documentations, including flask [14], django [11], matplotlib [41], pandas [52] and numpy [50]. We collect 215, 700, 4,089, 3,296 and 3,683 APIs from them, respectively.

(5) User-defined APIs. For code-based API recommendation, we regard all the functions defined in current projects as *user-defined APIs*. We do not explicitly collect them as a fixed set because they vary across projects. By inspecting the implementations, we can always identify the user-defined APIs.

3.2 Benchmark Datasets

In this section, we describe how we build the benchmark datasets APIBENCH-Q and APIBENCH-C.

3.2.1 *Creation of* APIBENCH-Q

We build the benchmark dataset APIBENCH-Q by mining Stack Overflow and tutorial websites. Note that we find that currently there is no query-based API recommendation approach specially designed for Python programs, but we still collect the query benchmark for it to facilitate further research investigation.

Mining Stack Overflow. As one of the most popular Q&A forums for developers, Stack overflow contains much discussion about the usage of APIs. Stack Overflow is the primary source for building APIBENCH-Q. We first download all posts from Aug 2008 to Feb 2021 on Stack Overflow (SO) via Stack Exchange Data Dump [13]. Each post is associated with a tag about the related programming language. We filter out the posts not tagged as Java or Python, resulting in 1,756,183 Java posts and 1,661,383 Python posts. Similar to other studies related to Stack Overflow mining [27], [55], we further filter out the posts based on the following rules:

> To increase the quality of the posts, we remove the posts that are not answered or do not have endorsed answers.
>
> We remove the posts that do not contain the HTML tag <code>, because we cannot extract any API from them.
>
> We remove the posts that contain code snippets longer than two lines, since we focus on single API recommendation in this paper and code snippets longer than two lines usually contain an API sequence. For multiple code snippets in one post, we remove the post only if all code snippets are longer than two lines.

We use string matching to find the APIs in the code of each post and remove the posts that do not contain any APIs involved in this paper, as described in Section 3.1.

After the rule-based filtering, we obtain 156,493 Python posts and 148,938 Java posts that contain descriptions about APIs. However, some of the posts are not directly related to API recommendation. For example, some posts only ask about comparing two similar APIs. The unrelated posts are hard to be automatically identified by rules. To ensure the relatedness of the posts in our benchmark dataset, we invite 16 participants with an average of 3-year development experience in Python or Java for manually checking. For each post, two of the participants are involved to check the following aspects:

1) whether the query asks about API recommendation;

2) whether the standard APIs recognized by the previous rules are intact, i.e., including the whole class and method names.

3) whether the APIs in answers exactly address the query.

If two participants provide the same answers for one post and also one of the above three aspects is not satisfied, we directly remove the post. If the two participants do not reach an agreement, the post will be forwarded to one of the authors to make a final decision.

As the remaining posts are still too many to be manually checked, we conduct two rounds of annotations. In the first round of annotations, we ask the annotators to label 100 randomly selected posts and conclude the reasons for the cases that they think are not about API recommendation. Then we collect the keywords that frequently appear in these unrelated cases, and remove the posts whose titles contain such keywords. For example, some post titles may contain some specific error names such as "AttributeError: 'Namespace' object has no attribute". We identify these titles and remove them because such posts tend to be related to debugging. However, we will keep the posts if the titles also contain the word "how", since we believe that the posts are likely to ask about error handling APIs. Although the filtering strategy is coarse, we can remove some noisy posts and facilitate manual annotation. In the second round annotations, we ask annotators to label all the remaining posts. It takes about one month to complete the two-round annotation process. After both rounds of annotations, we manually check 13,775 posts, in which 1,262 posts do not reach an agreement by the annotators and need further check by one of the authors. We use the commonly-used Fleiss Kappa score [15] to measure the agreement degree between the two annotators and the value is 0.77. The result indicates a high agreement between them. Based on the manual check, 3,245 of the 13,775 labeled posts remain. We take the titles of 3,245 posts as queries following previous studies [8], [27], and finally we get 1,925 Python queries and 1,320 Java queries. They comprise the first part of our benchmark APIBENCH-Q, as shown in the second column of Table 1.

Mining tutorial websites. API tutorial websites are the second major source of query-API pairs. We choose three popular API tutorial websites GeeksforGeeks [1], Java2s [2] and Kode Java [3] to establish APIBENCH-Q. Different from Stack Overflow that contains discussion on various topics, API tutorial websites focus on providing examples of how to use APIs. Therefore, manually annotating the relatedness of each query to API recommendation is not necessary. We adopt similar rules as mining Stack Overflow to filter out those without code snippets or associated with large code snippets. We finally collect 5,243 Java queries and 2,384 Python queries, which comprise the second part of our benchmark APIBENCH-Q, as shown in the fifth column of Table 1.

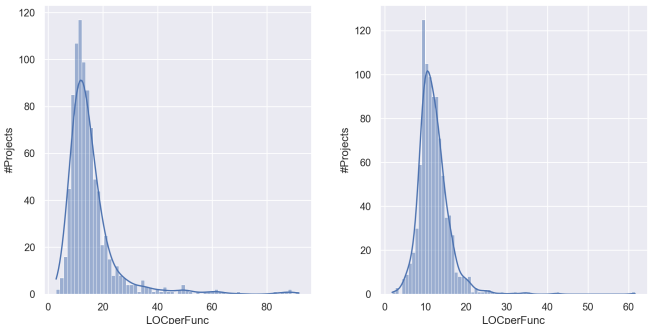


Fig. 3. Distribution of *code lines per function* for projects under *general* domain (Left: Python, Right: Java).

Note that we include all queries and corresponding APIs as our test set in APIBENCH-Q. We do not build a uniform training and validation set for query-based API recommendation approaches because the data sources used by current work are quite different. For example, using extra data sources is a major contribution for BIKER [27]. Lucene [16] does not need the training set at all. It is hard for us to build a unified training set for training all the approaches. To prevent potential data leakage, we remove the instances that overlap between training sets used by current approaches and APIBENCH-Q in preprocessing phase.

3.2.2 *Creation of* APIBENCH-C

We create the benchmark dataset APIBENCH-C by mining GitHub. GitHub [42] is one of the most popular websites for sharing code and includes large numbers of code repositories on different topics and programming languages.

In order to explore the performance of API recommendation under different domains, we first determine the domains for analysis. According to the JetBrains' developer survey and topic labels provided by GitHub[1] , we choose four popular domains for Python and Java, respectively, as shown in Table 2. For Python, we consider the domains "Machine Learning" (ML), "Security", "Web", and "Deep Learning" (DL); while for Java, we involve domains "Android", "Machine Learning" (ML), "Testing", and "Security". For each domain, we focus on the repositories tagged with the corresponding topic labels. For example, the "ML" domain only covers the repositories with the "machine learning" tag. As GitHub automatically aggregate all related projects under each domain, we directly collect 500 repositories with the most stars and 500 repositories with

1. https://github.com/topics

TABLE 3
The query reformulation techniques and query-based API recommendation approaches involved in the paper. For tools, the years they were last updated are listed. The column name "PL" indicates the applicable programming language.

Approach	Category/ Data Source	PL	Venue	Year
Query Reformulation				
Google Prediction Service [22]	Query expansion, modification	Any	-	2021
NLPAUG [40]	Query expansion, modification	Any	-	2021
SEQUER [8]	Query expansion, modification	Any	ICSE	2021
NLP2API [56]	Query expansion	Java	ICSME	2018
Query-Based API Recommendation				
RACK [58]	Official documentation, Stack Overflow	Java	ICSE	2016
KG-APISumm [38]	Official documentation, Wikipedia	Java	FSE	2019
Naive Baseline	Official documentation	Any	-	2021
DeepAPI [23]	Official documentation	Java	FSE	2016
Lucene [16]	Official documentation	Any	-	2021
BIKER [27]	Official documentation, Stack Overflow	Java	ASE	2018

TABLE 4
The code based API recommendation baselines included in this empirical study. For tools we list the year of its most recent update time. The column name "PL" indicates the applicable programming language.

Approach	Representation	PL	Venue	Year
Practical IDE				
PyCharm [30]	Code tokens	Python	-	2021
Visual Studio Code [43]	Code tokens	Python	-	2021
Eclipse [17]	Code tokens	Java	-	2021
IntelliJ IDEA [29]	Code tokens	Java	-	2021
Approach in Academia				
TravTrans [32]	AST	Python	ICSE	2021
PyART [24]	Token flow Data flow	Python	ICSE	2021
Deep3 [59]	AST, DSL	Python	ICML	2016
FOCUS [49]	API Matrix	Java	ICSE	2019
PAM [18]	API sequence	Java	FSE	2016
PAM-MAX	API sequence	Java	FSE	2016

the most forks on GitHub[2]. Besides the specific domains, we also build a "General" domain which only considers the popularity of repositories. For the "General" domain, we collect 1,000 repositories with the most stars and 1,000 repositories with the most forks on GitHub regardless of the topics.

Not all the collected repositories are applicable for code-based API recommendation. Some popular repositories do not contain enough code, e.g., only including documentations. To remove such repositories, we use *cloc* [5] to scan the code in each repository and filter out the repositories that 1) have fewer than 10 files or 2) have fewer than 1000 lines of code or 3) have code in Python or Java but with the ratio less than 10%. The number of projects, number of files, and average number of code lines for each domain of APIBENCH-C are shown in Table 2.

As most approaches [19], [24], [32], [49], [60] for code-based API recommendation require a training set to learn the API patterns or train the models, we split APIBENCH-C into a training set and a test set with a ratio of 80% and 20%, respectively. Note that we do not split a project both into the training set and test set, but put all the files of the same project into either the training set or test set, because Alon *et al.* [6] and LeClair *et al.* [36] find that code in the same project usually share the same variable names and code patterns, and splitting without considering project can cause data leakage. For the approaches requiring a validation set, we prepare it from the training set.

In order to study the impact of different recommendation points and different lengths of functions on the performance of current approaches, we analyze the average length of functions in each repository. we leverage Kernel Density Estimation (KDE) with Gaussian kernels to simulate the distributions. The distributions of the "General" domain for the Python and Java datasets are depicted in Figure 3. From the figure, we observe that function lengths are almost normally distributed. Most Python functions contain $5 \sim 30$ lines of code (LOC) and most Java functions contain $5 \sim 20$ lines of code. For studying the impact of function lengths, we divide the functions into extremely short functions, functions of moderate lengths, and extremely long functions according to the confidence interval under 90% confidence level. The confidence interval can be directly calculated by the standard deviations and means. We first determine the confidence interval of functions in different domains using standard deviations and regard the functions with lengths in the confidence interval as functions of moderate lengths. We regard functions with lengths smaller than the confidence interval as extremely short functions and functions with lengths larger than the confidence interval as extremely long functions. Note that except for the study on the impacts of function length, in other experiments we only consider functions of moderate lengths to guarantee that

2. The collection was conducted during April 2021.

our collected data is representative. The detailed thresholds of confidence intervals for distinguishing extremely long and short functions are illustrated in Table 2. We study the impact of function lengths on the performance of code-based approaches in Section 5.3.

In order to study the performance of current approaches on different kinds of APIs, we convert source files of each repository into ASTs and extract all the function calls in them. We label a function call as a *standard* API or *popular* third-party API if it matches one of the APIs collected in Sec. 3.1. We label a function call as a *user-defined* API if its implementation can be found in the current repository via import analysis. The average number of API calls per function, number of *standard* APIs and number of *user-defined* APIs are shown in columns 6 $\sim$ 8 of Table 2, respectively.

3.3 Implementation Details

In this section, we describe the details of each approach involved in the benchmark and the metrics for evaluation.

Query reformulation techniques. We choose four popular query reformulation techniques, including Google Prediction Service [22], NLPAUG [40], SEQUER [8], and NLP2API [56]. The detailed description of each technique is illustrated in Table 3. Google prediction service is included as one of the most effective approaches in practice, while SEQUER [9] is the state-of-the-art approach. NLPAUG [40] is considered since it is widely used for query reformulation in many NLP studies [31], [45], [54], [71]. We also include NLP2API [57] since it differs from major reformulation methods by first predicting the API class related to the query and then adding the predicted API class into the query.

Query-based API recommendation approaches. We choose five query-based API recommendation approaches published by recent top conferences, including KG-APISumm [38], BIKER [27], RACK [58], and DeepAPI [23], along with a popular search library Lucene [16]. The detailed description of each baseline is shown in Table 3. We reproduce the five approaches based on the replication packages released by the authors. Besides, we build a naive baseline that recommends APIs by computing the similarities between queries and API descriptions based on BERTOverflow [35]. The native baseline serves as an indicator of the basic performance of similarity-based models. We also notice that different sources are adopted by the approaches for creating the knowledge base. For example, the naive baseline and DeepAPI only consider official documentation, while BIKER and RACK also involve the Q&A forum – Stack Overflow. We list the knowledge source of each approach in Table 3. During implementation, we do not align the sources of the approaches, since the sources are claimed as contributions in the original papers. Instead, we design a separate RQ to study the impact of knowledge sources on the performance of API recommendations.

During studying the impact of query reformulation on the recommendation performance, we implement all the four query reformulation techniques for each of the six API recommendation baselines because all the baselines do not integrate query reformulation techniques in the original papers.

Code-based API recommendation approaches. We choose four IDEs and five approaches published on recent top conferences as our code-based API recommendation baselines. A detailed description of each baseline is shown in Table 4. For the IDEs and some of the approaches such as TravTrans [32] and Deep3 [59], they can predict any code tokens besides API tokens. In this paper, we focus on evaluating their performance in recommending APIs. Following prior research [19], [24], [32], [49], [60], we use the training set of APIBENCH-C to train each of the approaches in academia for a fair comparison.

PAM [19] is the only context-intensive approach, primarily designed for intra-project API pattern mining. In this paper, we also extend the approach to cross-project recommendation by selecting the best API from projects in the training set for each test case. The extended version of PAM is named as PAM-MAX, which indicates the theoretical maximum performance the context-insensitive approach can achieve.

Evaluation metrics. Since both query-based and code-based API recommendation baselines output a ranked list of candidate APIs, we adopt the commonly-used metrics in recommendation tasks for evaluation. The Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (NDCG) metrics are widely adopted by previous API recommendation studies [27]. In this study, we also involve a new metric Success Rate. The Success Rate@k is defined to evaluate the ability of an approach in recommending correct APIs based on the top-k returned results regardless of the orders. To determine the relevance score in NDCG calculation, we use a relevance score of 1 if an approach hits the correct API class, and a relevance score of 2 if the correct API method is hit. Therefore, we can align the performance of class-level and method-level approaches.

4 Empirical Results of Query Reformulation and Query Based API Recommendation

In this section, we study the RQ 1-3 discussed in Sec. 1 and provide the potential findings concluded from the empirical experiments. Since currently no query-based API recommendation approach is specially designed for Python APIs, we focus on studying query-based API recommendation approaches for Java.

4.1 Effectiveness of Query-Based API Recommendation Approaches (RQ1-1)

To answer RQ1, we evaluate the six query-based API recommendation baselines listed in Table 3 by using the original queries in our benchmark APIBENCH-Q. The evaluation results are illustrated in Table 5.

Class-level v.s. Method-level. Regarding the **class-level** recommendation, as shown in Table 5, we can find that BIKER achieves the highest Success Rate, e.g., 0.67 for Success Rate@10, indicating that BIKER is more effective in finding the correct API class in the top-10 returned results for 60%$\sim$70% of cases. Unsurprisingly, the naive baseline shows the worst performance for all the metrics. Even so, the naive baseline can successfully predict the correct API class for around 20% of cases. However, with respect to the **method-level** recommendation, all the approaches

TABLE 5
The basic performance of query-based API recommendation baselines without applying any query reformulation techniques at different metrics (Top-1,3,5,10). Note that we define NDCG as a uniform metric to evaluate class level and method level together, so the NDCG scores listed in two levels have the same values. The red numbers indicate the best performance achieved in top-10 results.

Baseline	Level	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
RACK	Class	0.17	0.30	0.35	0.41	0.17	0.23	0.24	0.24	0.25	0.17	0.24	0.26	0.28
KG-APISumm	Class	0.19	0.33	0.40	0.50	0.19	0.25	0.26	0.27	0.28	0.19	0.24	0.27	0.31
Naive Baseline	Class	0.07	0.13	0.16	0.21	0.07	0.10	0.10	0.10	0.11	0.07	0.09	0.10	0.13
	Method	0.02	0.03	0.04	0.05	0.01	0.02	0.03	0.03	0.03	0.07	0.09	0.10	0.13
DeepAPI	Class	0.19	0.27	0.29	0.30	0.19	0.22	0.23	0.23	0.23	0.17	0.22	0.23	0.24
	Method	0.05	0.09	0.10	0.11	0.05	0.07	0.07	0.07	0.07	0.17	0.22	0.23	0.24
Lucene	Class	0.15	0.21	0.24	0.29	0.15	0.17	0.18	0.17	0.19	0.12	0.15	0.16	0.20
	Method	0.04	0.08	0.10	0.14	0.04	0.06	0.06	0.06	0.07	0.12	0.15	0.16	0.20
BIKER	Class	0.33	0.51	0.59	**0.67**	0.33	0.41	0.41	**0.39**	**0.44**	0.27	0.32	0.35	**0.42**
	Method	0.12	0.23	0.29	**0.37**	0.12	0.16	0.18	**0.18**	**0.19**	0.27	0.32	0.35	**0.42**

show obvious declines. For example, the Success Rate@10 of BIKER is only 0.37, decreasing by 44.8% compared to the class-level recommendation. The Success Rates@10 of DeepAPI and Lucene are only around 0.10, which is far from the requirement of practical development. On average, the approaches fail to give the exact methods for 57.8% APIs that they give the correct classes in top-10 returned recommendations. Thus, recommending method-level APIs still remains a great challenge.

Finding 1: *Existing approaches fail to predict 57.8% method-level APIs that could be successfully predicted at the class level. The performance achieved by the approaches is far from the requirement of practical usage. Accurately recommending the method-level APIs still remains a great challenge.*

Retrieval-based methods v.s. Learning-based methods. By comparing learning-based methods, such as DeepAPI and naive baseline, with the other retrieval-based methods, we can observe that learning-based methods achieve relatively lower performance regarding the Success Rate@10 metric. For example, on average, retrieval-based methods can accurately predict 46.8% class-level and 25.5% method-level APIs among all the cases in the top-10 returned results, respectively, while learning-based methods can only successfully recommend 25.5% class-level and 8% method-level APIs. A possible reason may be the insufficient training data for the learning-based methods in this task domain. Since there are more than 30,000 APIs from the official documentation, learning-based methods require a large number of query-API pairs for training. However, even the largest Q&A forum, Stack Overflow, contains only about 150,000 posts after our pre-processing, which is not enough for model training.

Finding 2: *Learning-based methods do not necessarily outperform retrieval-based methods in recommending more correct APIs. The insufficient query-API pairs for training limit the performance of learning-based methods.*

Performance in API ranking. From Table 5, we find that there exist obvious gaps between the scores of Success Rate@k and the metrics for evaluating API ranking, such as MAP@k and NDCG@k. For example, RACK achieves Success Rate@10 score at 0.41, but its MAP@10 score is only 0.24. This indicates that although the approaches are able to find the correct APIs, they cannot well rank them ahead in the returned results. The low MRR scores, e.g., 0.11 $\sim$ 0.44 for class-level API recommendation and 0.03 $\sim$ 0.19 for method-level API recommendation, and NDCG scores also show the poor ranking performance of the approaches. The results manifest that API ranking is still challenging for current approaches.

Finding 3: *Current approaches cannot rank the correct APIs well, considering the huge gap between the scores of Success Rate and the other ranking metrics.*

To sum up, accurately recommending method-level APIs and ranking candidate APIs still remain great challenges. Besides, the insufficient data for training hinder the performance of current learning-based approaches.

4.2 Effectiveness of Query Reformulation Techniques (RQ2)

Original queries can be short in length or contain vague terms. Query reformulation aims at changing original queries for facilitating downstream tasks. In this RQ, we explore the impact of query reformulation on the performance of query-based API recommendation.

We implement the four query reformulation techniques, as listed in Table 3, for the original queries. We name the queries reformulated by query expansion techniques and query modification techniques as expanded queries and modified queries, respectively. For each original query, we conduct the reformulation 10 times, producing 10 expanded or modified queries, with the statistics shown in Table 1. Note that NLPAUG [40] is a comprehensive data augmentation library for general NLP tasks. We choose the popular word-level insertion and substitution methods designed for manipulating single sentences based on five models, including BERTOverflow [35], Google News Word2vec [21],

Stack Overflow Word2vec [68], WordNet [44], and Random model, in the library to generate expanded and modified queries.

The queries output by the query reformulation techniques are not ranked in order, and may impact the downstream API recommendation performance variously. To explore the maximum potential effect brought by query reformulation techniques, we evaluate the API recommendation approaches on each reformulated query and choose the best result for analysis. We choose the maximum improvement instead of average improvement for analysis based on the following considerations: 1) It is hard to provide a fair comparison between query reformulation approaches that rank the processed queries such as SEQUER and query reformulation approaches that do not rank the processed queries such as NLPAUG. 2) Query modification would change the query semantics [8], [37]; therefore, using average improvement tends to involve wrong queries and bias the evaluation results. 3) Our goal is to show the potential of current query reformulation approaches, and motivate future research on query reformulation to enhance the performance of API recommendation.

We study the impact of query reformulation on API recommendation from the following two aspects:

1) whether query reformulation techniques can help predict more correct APIs;

2) whether query reformulation can improve the API ranking performance.

4.2.1 Influence on predicting more correct APIs

With query reformulation v.s. Without query reformulation. The Success Rate metric reflects the proportion of the APIs an approach can correctly predict. The results of implementing the reformulation techniques on API recommendation approaches are illustrated in Figure 4 (class-level) and Figure 5 (method-level). From the figures, we observe that query reformulation can increase the performance of API recommendation in most cases. Only for a few cases, the performance drops, which can be attributed to the inefficiency of some query reformulation techniques. For example, NLPAUG (WordNet) and NLPAUG (Random) tend to poorly modify the original queries for recommendation, as shown in Figure 4 (b) and Figure 5 (b). Overall, on average the process improves the class-level and method-level recommendation by 0.11 and 0.08, which is a corresponding boost of 27.7% and 49.2% compared with the basic performance on original queries.

Finding 4: *Query reformulation techniques are quite effective in helping query-based API recommendation approaches give the correct API by adding an average boost of 27.7% and 49.2% on class-level and method-level recommendations, respectively.*

Query expansion v.s. Query modification. By comparing the class-level and method-level recommendation results of query expansion and query modification in Figure 4 and Figure 5, respectively, we observe that all the query expansion techniques improve the API recommendation performance, but not all the query modification techniques benefit the recommendation. For example, NLPAUG (WordNet) and NLPAUG (Random) generally decrease the performance of current approaches both in class-level and method-level recommendations. This indicates that query expansion techniques bring more stable improvement than query modification techniques. Furthermore, on average, query expansion techniques improve the performance by 0.13 and 0.10 on class-level and method-level recommendation, which is much higher than the improvement of 0.09 and 0.06 achieved by query modification techniques. This also suggests that query expansion techniques are more effective than query modification techniques.

Finding 5: *Query expansion is more stable and effective to help current query-based API recommendation approaches give correct APIs than query modification.*

Comparing different query expansion techniques. As shown in Figure 4 (a) and Figure 5 (b), NLP2API and NLPAUG (BERT) present the largest improvement on the performance of query-based API approaches at both class level and method level. For analyzing the improvement, we use two examples to illustrate the query expansion results of NLP2API and NLPAUG (BERT), respectively. In both examples, the most effective approach BIKER fails to predict the API based on the original queries but succeeds given the reformulated queries.

Example 1: Query Expansion	
TECHNIQUE	NLP2API
ORIGINAL QUERY	Returns a new Document instance
PROCESSED QUERY	DocumentBuilderFactory Returns a new Document instance

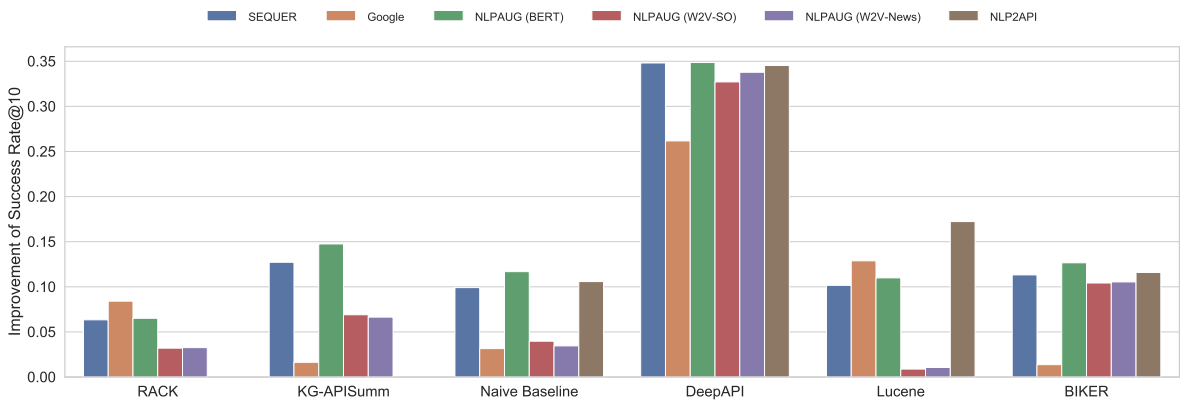
In the first example, NLP2API expands the query by adding a predicted API class *DocumentBuilderFactory* that is related to the original query. With such an explicit hint, the recommendation approach can narrow down the search scope and pinpoint the requested API method.

Example 2: Query Expansion	
TECHNIQUE	NLPAUG (BERT)
ORIGINAL QUERY	Java reverse string
PROCESSED QUERY	java reverse character string

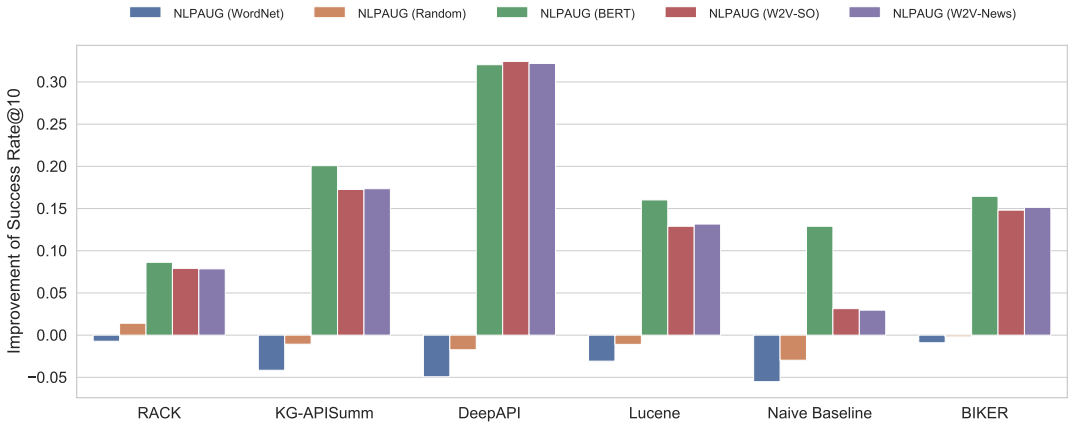
In the second example, the query is looking for the API *java.lang.StringBuilder.reverse()*, whose description in official documentation is "*Causes this character sequence to be replaced by the reverse of the sequence*". NLPAUG (BERT) adds a relevant word *character* to enrich the semantics of the original query.

Example 3: Query Expansion	
TECHNIQUE	NLPAUG (W2V-SO)
ORIGINAL QUERY	Convert from Radians to Degrees in Java
PROCESSED QUERY	Convert from AV Radians to Degrees in Long Java

Comparing NLPAUG (W2V) with NLPAUG (BERT) and NLP2API in Figure 4 and Figure 5, we find that the NLPAUG (W2V) is much less effective. To obtain a possible reason for such a difference, we give the third example below. As shown in this example, we find that NLPAUG



(a) Query Expansion



(b) Query Modification

Fig. 4. The maximum improvement of Success Rate@10 by all query reformulation techniques on **class-level** query-based API recommendation baselines. We do not evaluate the performance of RACK and KG-APISumm in NLP2API reformulated queries as they are only class-level recommendation approaches while NLP2API directly give the predicted API classes. Note that we include Google Prediction Service and SEQUER as expansion techniques here because they expand the queries in most cases.

(W2V) adds two irrelevant words into the original query, which negatively impacts the prediction results of BIKER. This also indicates that contextual embeddings such as BERT are more effective than traditional word embeddings.

Finding 6: *In query expansion, adding predicted API class names or relevant words to queries are more useful than adding other tokens.*

Comparing different query modification techniques. Among all query modification techniques, NLPAUG (BERT) presents the biggest improvement on all the baselines at both class level and method level. Example 4 illustrates how NLPAUG (BERT) modifies words in the original query. In the example, the original query asks about ways to calculate the time difference between two dates and the correct API is *java.time.Period.between()*. The description of the API in its official documentation is "*obtains a period consisting of the number of years, months, and days between two dates*". However, the word "*difference*" used in the original query does not clearly describe the functional request. NLPAUG (BERT) modifies the word into "*months*" which exactly appears in the official description. Based on the modifications, the correct API is recommended.

From the second and fourth examples above, we find

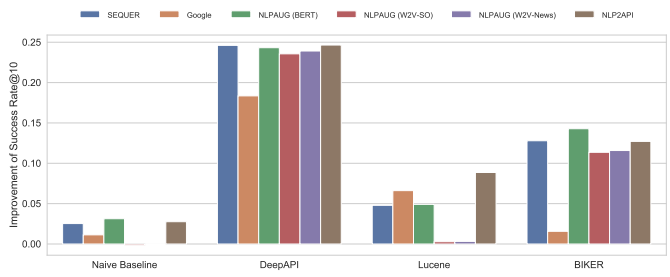
Example 4: Query Modification	
TECHNIQUE	NLPAUG (BERT)
ORIGINAL QUERY	How do I calculate difference between two dates
PROCESSED QUERY	how do they ~~I~~ calculate months ~~difference~~ between two dates

that BERT-based models show great performance on both query expansion and query modification to help improve the performance of current query-based API recommendation approaches. This indicates that even though the current data source limits the performance of these models to directly predict the correct APIs, they can be used to improve the query quality as query reformulation techniques.

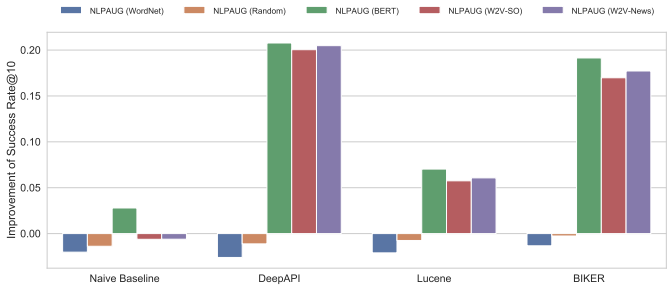
Finding 7: *BERT-based data augmentation shows superior performance in query modification compared with other query modification techniques.*

4.2.2 *Influence on the performance of API ranking*

In this section, we analyze the impact of query reformulation techniques on the performance of API ranking. Since the ideal case is that the correct APIs rank first in the



(a) Query Expansion



(b) Query Modification

Fig. 5. The maximum improvement of Success Rate@10 by all query reformulation techniques on **method-level** query-based API recommendation baselines.

returned results, we use the metric NDCG@1 which considers both class-level and method-level recommendation performance. We compute the changes of NDCG@1 scores for the query-based API recommendation approaches before and after query reformulation. Besides, to focus our analysis on the performance of API ranking instead of the overall recommendation accuracy, the computation is performed only on the cases that are correctly predicted with and without query reformulation.

The results are illustrated in Figure 6. As can be seen in Figure 6 (a), most query expansion techniques also improve the ranking results of the query-based recommendation approaches. Among all the query expansion techniques, SEQUER, NLPAUG (BERT), RACK and NLP2API can improve the ordering performance relatively better than the others. The biggest improvement of 0.14 (32% boost) is achieved by NLP2API on the Lucene approach. We also find that on average query expansion also improves MRR by 0.09 (36% boost) and 0.08 (89% boost) on class-level and method-level recommendation, respectively, which indicates that the correct APIs are ranked much higher based on reformulated queries.

According to Figure 6 (b), compared with query expansion techniques, query modification techniques are much less effective in improving the API ranking performance. For example, the average improvement of NDCG@1 brought by query modification is 0.01 (4% boost), which is 0.06 (14% boost) for query expansion techniques. Comparing different data augmentation methods, we also find that WordNet and random methods tend to negatively impact the ranking results, leading to 24% and 14% drop in terms of NDCG@1, respectively. The results indicate that inappropriate query modification will reduce the ranking performance of the query-based recommendation approaches.

Finding 8: *Expanding queries or modifying queries with appropriate data augmentation methods can improve the ranking performance of the query-based API recommendation techniques.*

To sum up, query reformulation, especially query expansion, can not only help current approaches recommend more correct APIs, but also improve the ranking performance. However, the reformulation step is generally ignored by current studies. Future work is suggested to involve such a step for more accurate API recommendation.

4.2.3 A special Query Modification Method: Word Deletion

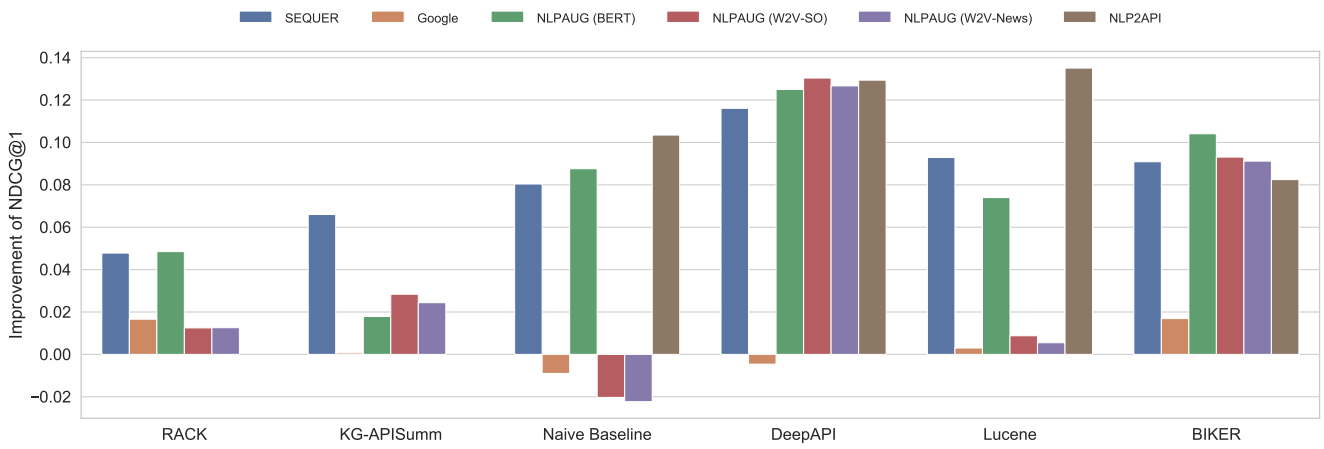
In previous subsections, we compare and evaluate different query expansion and modification techniques. They aim at enriching the original queries by adding, replacing or modifying some words without deleting words. In this section, we focus on studying the impact of word deletion, a special query modification method, on the performance of query-based API recommendation approaches. Different from the previous query reformulation techniques which rely on external data sources, the word deletion method we studied does not leverage any extra knowledge. Our goal is to explore whether original queries contain meaningless or noisy words. Specifically, we randomly delete some words from the original query every time and produce ten different modified queries for one original query.

The maximum and average Success Rate@10 scores based on the modified queries are illustrated in Figure 7. As can be seen, the average performance of the query-based API recommendation approaches, denoted as the orange bar, decreases by 0.05 (13% drop) at class level and 0.03 (18% drop) at method level. The results are not surprising, and indicate that most words in the original queries are helpful for the recommendation. However, the maximum scores, denoted as the green bar, all show that word deletion improves the recommendation performance with an average boost of 38% and 64% for class level and method level, respectively. The improvement demonstrates that the original queries contain noisy words that can bias the recommendation results, although most of the words are useful for recommendation.

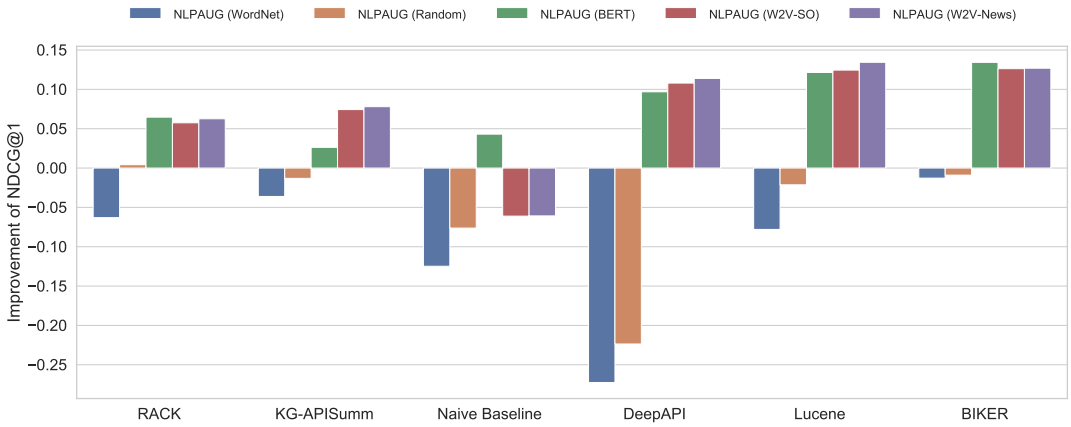
After checking all cases, we find that word deletion is helpful for successfully recommending APIs of 545 queries, which maybe attributed to that some noisy words are removed from the original queries. To understand what kinds of words are noisy for the accurate recommendation of these 545 queries, we manually compare the original queries and processed queries. We summarize three possible situations as below:

1) 349 (64%) queries contain unnecessary or meaningless words.

Example 5: Word Deletion	
TECHNIQUE	Random Deletion
ORIGINAL QUERY	Standard way to iterate over a StringBuilder in java
PROCESSED QUERY	~~Standard way to~~ iterate over a StringBuilder ~~in java~~



(a) Query Expansion



(b) Query Modification

Fig. 6. The maximum improvement of NDCG@1 by all query reformulation techniques on query-based API recommendation baselines under original successful cases.

In Example 5, the phrases "*Standard way to*" and "*in java*" are not beneficial for pinpointing the correct API. Stop word removal also has a limited effect on eliminating these words.

2) 156 (29%) queries contain too detailed words for explanation.

Example 6: Word Deletion	
TECHNIQUE	Random Deletion
ORIGINAL QUERY	converts a color into a string like 255,0,0
PROCESSED QUERY	converts a color into a string ~~like 255,0,0~~

In Example 6, the phrase "*like 255,0,0*" is used to explain the "*string*". However, such phrases never appear in the official documentation and the specific number adversely impacts the recommendation results.

3) 34 (6%) queries contain extreme long descriptions.

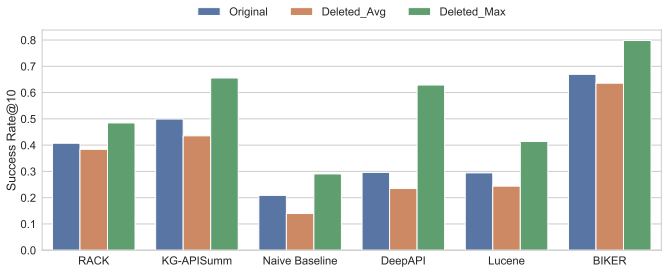
Example 7: Word Deletion	
TECHNIQUE	Random Deletion
ORIGINAL QUERY	how to add progress bar to zip utility while zipping or extracting in java
PROCESSED QUERY	how to add progress bar to zip utility ~~while zipping or extracting in java~~

Based on work [8], most queries have the lengths of between one to seven words. In our manual analysis process, one query is regarded as extremely long if it contains more than 10 words. In Example 7, the words after "*while*" actually describe nothing about the task. The long query descriptions can decrease the weight of useful words in the queries thus confusing API recommendation approaches.

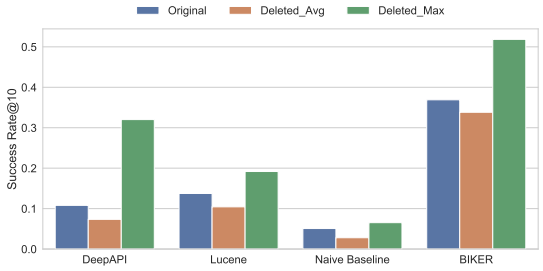
Finding 9: *Original queries raised by users usually contain noisy words which can bias the recommendation results, and query reformulation techniques should consider involving noisy-word deletion for a more accurate recommendation.*

4.3 Data Sources (RQ3)

In RQ1-1, we highlight that insufficient data greatly limits the performance of current learning-based methods. In this section, we conduct a deep analysis on the influence of different data sources on the recommendation results. From Table 3, we can observe that current approaches generally leverage three different data sources: official documentation, Q&A forums, and tutorial websites. For analysis, we choose two methods, Lucene and naive baseline, which are flexible to incorporate different data sources. Specifically, we evaluate the methods on the part of queries from the



(a) Class Level



(b) Method Level

Fig. 7. The maximum and average Success Rate@10 on all baselines when randomly deleting some words in original queries.

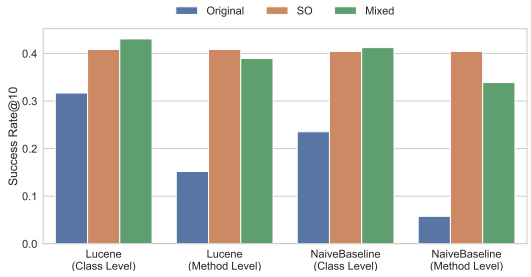


Fig. 8. The Success Rate@10 of Lucene and Naive Baseline under three data source settings.

tutorial websites collected in APIBENCH-Q, and the method training is conducted based on the following knowledge base:

1) only official documentation,
2) only Stack Overflow posts, and
3) both official documentation and Stack Overflow posts.

The experiment results are shown in Figure 8. As can be seen, training on Stack Overflow posts achieves much better performance than on official documentation at both class and method levels. For example, Lucene achieves a 29% boost in class-level and an 169% boost in method-level recommendation when searching based on Stack Overflow than on official documentation; and the naive baseline even achieves a 71% boost in class-level and a 602% boost in method-level recommendation. The advantage of leveraging Stack Overflow posts may be attributed that the discussion on Stack Overflow is more natural and similar to user queries, compared with the descriptions in the official documentation. Besides, the extended usage of some APIs is rarely mentioned in official documentation but is widely discussed in Stack Overflow. An example is used to illustrate the influence of different data sources.

In Example 8, the query asks about the API for generating an MD5 hash of a file. However, there is no standard API specially designed to generate the MD5 hash, so Lucene focuses on two words "*hash*" and "*file*" for recommendation. But the official description of ground truth API *java.security.MessageDigest.digest()* does not contain the word "*file*" since it is a general API that not merely handles files. Under this circumstance, Lucene recommends a more relevant but wrong API *java.nio.file.attribute.FileTime.hashCode()*. When involving Stack overflow posts, as there already exists discussion on how to generate the MD5 hash, Lucene can easily pinpoint and recommend the correct API in the posts.

Example 8: Data Source

BASELINE	Lucene
ORIGINAL QUERY	Compute the md5 hash of a File
CORRECT API	java.security.MessageDigest.digest(), java.security.MessageDigest.getInstance()
API DESCRIPTION	Completes the hash computation by performing final operations such as padding
SIMILAR SO POST	How can I generate an MD5 hash in Java?

The advantage of leveraging Stack Overflow for recommendation is also demonstrated by the BIKER approach [27], which is the most effective approach in Section 4.1. Our finding is consistent with the claim in the work [27] that Stack Overflow posts can mitigate the semantics gap between user queries and official descriptions.

Finding 10: *Apart from official documentation, using other data sources such as Stack Overflow can significantly improve the performance of query-based API recommendation approaches.*

5 EMPIRICAL RESULTS OF CODE-BASED API RECOMMENDATION

In this section, we study the RQ1 and RQ 4 $\sim$ 6 discussed in Sec 1. To study RQ1, RQ4 and RQ5, we evaluate the performance of all the code-based API recommendation approaches on the "General" domain of our benchmark APIBENCH-C, as shown in Table 2, since the "General" domain includes code with different topics and can reflect the overall performance of baselines. For studying the ability of cross-domain adaptation in RQ6, we evaluate the performance of the approaches on all the five domains of our APIBENCH-C.

5.1 Effectiveness of Existing Approaches (RQ1-2)

According to Table 4, three approaches for Python and three approaches for Java are evaluated on the "General" domain of APIBENCH-C. The results are depicted in Table 6. We can observe that the learning-based method TravTrans obtains the best performance on the Python dataset, achieving 0.62 and 0.54 for Success Rate@10 and NDCG@10, respectively. The results mean that TravTrans can successfully recommend 62% of APIs in our benchmark and well predict the API rankings. However, the traditional statistical method Deep3 only achieves 0.43 and 0.32 for Success Rate@10 and NDCG@10, respectively, while the pattern-based method FOCUS and PAM achieve less than 0.10 for both Success

TABLE 6
The performance of code-based API recommendation baselines at different metrics (Top-1,3,5,10). All baselines are trained and tested on the full dataset from "General" domain of APIBENCH-C except for PyART. Since PyART takes months to train and test on our full dataset, we randomly sampled 20% of original training and testing testset to evaluate it. The "PL" column indicates the programming language the baselines target. The red number indicates the best performance.

PL	Baseline	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
Python	TravTrans	0.45	0.57	0.59	**0.62**	0.45	0.50	0.51	**0.51**	**0.51**	0.45	0.52	0.53	**0.54**
	Deep3	0.21	0.34	0.37	0.43	0.20	0.27	0.28	0.28	0.28	0.21	0.29	0.30	0.32
	PyART	0.29	0.38	0.46	0.60	0.29	0.33	0.35	0.37	0.37	0.29	0.34	0.37	0.41
Java	FOCUS	0.01	0.03	0.04	0.06	0.01	0.02	0.02	0.03	0.03	0.01	0.02	0.03	0.04
	PAM	0.01	0.02	0.03	0.05	0.01	0.02	0.02	0.02	0.02	0.01	0.02	0.02	0.03
	PAM-MAX	0.22	0.32	0.36	**0.45**	0.22	0.26	0.27	**0.28**	**0.28**	0.22	0.27	0.29	**0.32**

TABLE 7
The performance of code-based API recommendation baselines along with 4 widely used IDEs tested on 500 cases sampled from the testset of all domains in APIBENCH-C. The "PL" column indicates the programming language the baselines target. The red number indicates the best performance. The rows with gray background indicates the performance of IDEs.

PL	Baseline	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
Python	TravTrans	0.38	0.46	0.48	**0.50**	0.38	0.42	0.42	**0.43**	**0.43**	0.38	0.43	0.44	**0.44**
	Deep3	0.19	0.26	0.31	0.38	0.19	0.22	0.23	0.24	0.24	0.19	0.23	0.25	0.28
	PyCharm	0.31	0.42	0.47	0.49	0.31	0.36	0.37	0.37	0.37	0.31	0.38	0.40	0.40
	VSCode	0.05	0.15	0.21	0.35	0.05	0.09	0.11	0.13	0.13	0.05	0.11	0.14	0.18
Java	FOCUS	0.02	0.04	0.05	0.07	0.02	0.03	0.03	0.04	0.04	0.02	0.03	0.04	0.04
	PAM	0.01	0.02	0.05	0.07	0.01	0.02	0.02	0.03	0.03	0.01	0.02	0.03	0.04
	PAM-MAX	0.27	0.38	0.43	0.56	0.27	0.31	0.33	0.34	0.34	0.27	0.33	0.35	0.39
	Eclipse	0.28	0.42	0.49	0.60	0.28	0.34	0.35	0.37	0.37	0.28	0.36	0.39	0.42
	IntelliJ IDEA	0.42	0.58	0.65	**0.67**	0.42	0.49	0.51	**0.51**	**0.51**	0.42	0.51	0.54	**0.55**

Rate@10 and NDCG@10. This suggests that learning-based methods obtain superior performance in code-based API recommendation, which is quite different from query-based API recommendation. The possible reason is that lots of well-organized public code repositories provide sufficient data for training code-based API recommendation models.

We also find that FOCUS and PAM show low recommendation accuracy, with all the metric values lower than 0.1. The low performance is attributed to the context representation of the approaches. PAM is a context-insensitive approach, which only mines the top-N APIs that are most likely to be used in the training set and directly recommends them for each file in the test set; while FOCUS takes one step further by extracting the APIs in the test set and building a matrix to match the APIs in the training set. Such coarse-grained context representation or context-insensitive representation does not well capture the relations between APIs. PAM-MAX shows the theoretical best performance context-insensitive methods can achieve. However, the performance of PAM-MAX is still lower than that of TravTrans and PyART which consider fine-grained code features such as code tokens and data flows. The results indicate the effectiveness of fine-grained approaches for code-based API recommendation.

Besides the recent code-based API recommendation approaches, we also compare the widely-used IDEs. Since it is hard to automatically evaluate IDEs' recommendation performance, we sampled 500 APIs from the original large test set of APIBENCH-C based on the distribution shown in Table 2. We then conduct a manual evaluation by imitating the behaviors of developers on the 500 sampled APIs. We show the results on the sampled test set in Table 7. As can be seen, for Python, Pycharm achieves the Success rate@10 at 0.49 and NDCG@10 at 0.40, which is truly competitive to the performance of TravTrans, with Success Rate@10 and NDCG@10 at 0.50 and 0.44, respectively. For Java, IDEs also show competitive performance compared with the baseline approaches. The results demonstrate that the widely-used IDEs are generally effective in API recommendation and far from relying on alphabet orders for recommendation.

Finding 11: *DL models such as TravTrans show superior performance on code-based API recommendation by achieving a Success Rate@10 of 0.62, while widely-used IDEs also obtain satisfying performance by achieving a Success Rate@10 of 0.5 $\sim$ 0.6.*

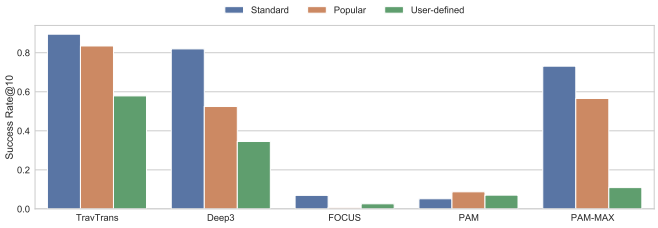


Fig. 9. The Success Rate@10 of baselines on three categories of APIs at the "General" domain of APIBENCH-C.

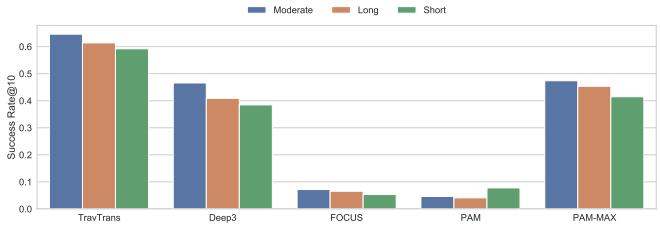


Fig. 10. The Success Rate@10 of baselines on extremely short, normal and extremely long contexts at the "General" domain of APIBENCH-C.

5.2 Capability to Recommend Different Kinds of APIs (RQ4)

Exploring which kinds of APIs tend to be wrongly predicted is essential for understanding the bottleneck of current approaches and for providing clues for further improvement. In Section 3, we have classified all APIs into standard APIs, popular third-party APIs and user-defined APIs. In this RQ, we study the performance of current baselines for different kinds of APIs. Specifically, we evaluate TravTrans, Deep3, FOCUS, PAM and PAM-MAX on the full test set of the "General" domain, with results shown in Figure 9.

As can be seen in Figure 9, most approaches achieve a very high Success Rate@10 on standard APIs. For example, TravTrans even successfully recommends more than 90% of standard APIs in the test set. The approaches also present relatively good performance for the popular third-party libraries, e.g., TravTrans achieves a Success Rate@10 of more than 0.8. As standard APIs and popular APIs from third-party libraries are widely used in real-world projects, data-driven methods can achieve superior performance. However, it is hard for the approaches to correctly recommend the user-defined APIs as they fail to predict 35.3% $\sim$ 91.3% more of user-defined APIs comparing to the prediction of standard APIs.

Finding 12: *Although current approaches achieve good performance on recommending standard and popular third-party libraries, they face the challenges of correctly predicting the user-defined APIs.*

5.3 Capability to Handle Different Contexts (RQ5)

As context representation is an important part of the current code-based API recommendation shown in Figure 2, it is worthwhile to study the impact of different contexts on the performance of current approaches. In this RQ, we explore the impact of the following two different types of context.

lengths of functions, which evaluates the capability of current approaches to handle different lengths of contexts;
different recommendation points, since different recommendation points affect how much context an approach can be aware of before recommendation.

Capability to handle different lengths of functions. In Section 3 and Table 2 we classify all functions of APIBENCH-C into extremely short functions, functions of moderate lengths, or extremely long functions by sampling the first 5%, middle 90% and last 5% according to the distribution of function lengths. As code-based API recommendation is often based on the context in a function, the length of function can represent the length of context that an approach needs to handle. We study the performance of current baselines on functions of different lengths and show the results of TravTrans, Deep3, FOCUS, PAM, and PAM-MAX in Figure 10.

From Figure 10, we find that most baselines share similar performance distributions on functions of different lengths. They present the best performance on functions with moderate lengths and suffer from performance drops on extremely long or short functions. To be more specific, the performance drops by 7.1% for extremely long functions and 10.6% for extremely short functions on average. The results indicate that context length can affect the performance of current approaches. Besides, the approaches are more difficult to recommend correct APIs for the functions of extremely short lengths than those of extremely long lengths.

Finding 13: *Context length can impact the performance of current approaches in API recommendation. The approaches perform poorly for the functions with extremely short or long lengths, and accurate recommendation for the extremely short functions is more challenging.*

Capability to handle different recommendation points. Similar to the previous work [49], we first define three locations of recommendation points. Suppose that the LOC of a function is n and the total number of APIs used in the function is m. we define a recommendation point that is on the a_{th} line of the function and is the b_{th} API in the function locates on

1) the front of function if a=n < 1=4 and b=m < 1=4, or
2) the middle of function if 1=4 < a=n < 3=4 and 1=4 < b=m < 3=4, or
3) the back of function if a=n > 3=4 and b=m > 3=4.

We show an example for illustrating front, middle and back recommendation point in listing 1.

```
public static void main(String args[]) {
//first 1/4 part
    String[] strArray =
  new <Front Recommendation Point>
...
//middle 1/2 part
    List l = Arrays.<Middle Recommendation Point>
...
//last 1/4 part
    Collections.<Back Recommendation Point>;
...
  }
```

Listing 1. Example of Recommendation Points

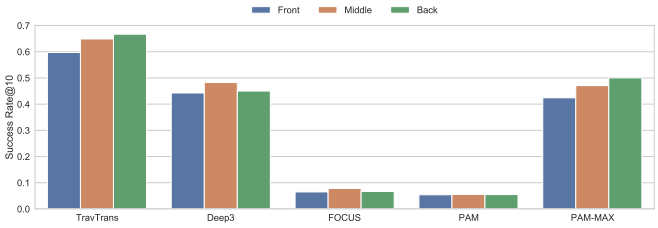


Fig. 11. The Success Rate@10 of baselines on three categories of recommendation points at the *general* domain of APIBENCH-C.

For all the APIs in the test set of the "General" domain, we replace them with placeholders of the above three types of recommendation points for evaluation. We also remove APIs in extremely long or short functions (according to the thresholds shown in Table 2) to alleviate the influence of function lengths. We show the results of TravTrans, Deep3, FOCUS, PAM and PAM-MAX in Figure 11.

From Figure 11, we observe that current approaches generally perform worse at the front recommendation points by achieving an average Success Rate@10 of 0.316. This is intuitive since there exists less information for current approaches to leverage at front recommendation points. However, it is worth noting that not all approaches achieve the best performance at the back recommendation point which is associated with the most context among all the recommendation points. The reason may be that the approaches cannot well handle the overwhelming information in long contexts.

Finding 14: *The location of recommendation points can affect the performance of current approaches. Current approaches perform worst at front recommendation points due to limited contexts. Some of them also suffer from overwhelming contexts at back recommendation point.*

To sum up, different contexts can affect the performance of current code-based API recommendation approaches. Among them, the extremely short contexts and front recommendation points bring the most challenges for the accurate recommendation.

5.4 Adaptation to Cross-Domain Projects (RQ6)

We have divided APIBENCH-C into five different domains in Section 3. In this section, we aim at studying the adaption capability of current approaches for cross-domain projects. We train the approaches in one domain and evaluate them in other different domains. We choose the approaches TravTrans, Deep3, and PyART, which are all designed for Python, for analysis. We do not involve the approaches FOCUS, PAM, or PAM-MAX, since they use coarse-grained context representations or context-insensitive feature, and are difficult to incorporate project-specific information. The first four rows of Table 8 list the cross-domain Success Rate@10 of TravTrans, Deep3 and PyART, respectively.

According to Table 8, the approaches trained on one domain generally perform best on the test set of the same domain. For example, when trained on data from the "Security" domain, TravTrans, Deep3 and PyART obtain the best scores at 0.54, 0.51 and 0.48 on the test set of the same domain, respectively, in terms of Success Rate@10. However, their performance drops by 2.1% $\sim$ 43.1% when recommending APIs from different domains.

Finding 15: *Current approaches using fine-grained context representation are sensitive to the domain of the training data and suffer from performance drop when recommending cross-domain APIs.*

We also analyze the cross-domain performance of the approaches when training on multiple domains instead of on one single domain. Such analysis is worthwhile to explore whether different domains can complement each other. Then we train the approaches on the projects from the "General" domain of APIBENCH-C and evaluate them on the other four different domains. We show the results in the last row of table 8.

From the table, we can see that the approaches trained on the "General" domain generally show the best performance when evaluating on different domains. For example, TravTrans trained on the "General" domain achieves the Success Rate@10 of 0.72, 0.76, 0.78 and 0.74 on ML, Security, Web and DL domains, respectively, which is significantly higher than the corresponding best scores obtained by TravTrans trained on single one domain. We observe an average boost of 14% for the performance of the approaches when trained on multiple domains than on a single domain. The results indicate that training approaches on multiple domains greatly improve the recommendation performance.

Finding 16: *Training on multiple domains helps the current approaches to recommend APIs in different single domains, and the performance is generally better than only training on a single domain.*

6 Discussion and Future Work

6.1 Query Reformulation for Query-based API Recommendation

In Section 4.2, we find that query reformulation techniques can not only help current query-based API recommendation approaches find more correct APIs but also improve the ranking performance. Based on query reformulation, BIKER can even achieve a Success Rate@10 of 0.80 in class-level and 0.51 in method-level API recommendation. The results demonstrate that query quality has a great impact on the recommendation results and suggest that query reformulation should become a common pre-processing technique used before query-based API recommendation. We also discover that some query reformulation techniques, such as adding predicted API class names or relevant words, can improve the performance of query-based API recommendation approaches. However, to the best of our knowledge, few studies have considered integrating these techniques, which could be one major reason that current approaches achieve limited performance.

By implementing a random deletion strategy, in Section 4.2.3 we find that user queries usually contain noisy words, which can bias the recommendation results. We summarize three kinds of cases in which a query contains noisy words. However, there exists very little work that

TABLE 8
The cross-domain Success Rate@10 of Python code-based API recommendation baselines. The rows list the domains where three baselines are trained and the columns list the domains where three baselines are evaluated. The red number indicates the best performance an approach achieves when trained on one domain (The largest number in each row). The numbers with gray background indicates the best performance achieved on a specific testing domain (The largest number in each column).

Training Domain	TravTrans				Deep3				PyART			
	ML	Security	Web	DL	ML	Security	Web	DL	ML	Security	Web	DL
ML	0.64	0.58	0.53	**0.71**	0.42	0.41	0.36	**0.48**	0.39	0.35	0.40	**0.40**
Security	0.40	**0.54**	0.54	0.39	0.31	**0.51**	0.42	0.29	0.36	**0.48**	0.47	0.36
Web	0.54	0.63	**0.64**	0.51	0.33	0.42	**0.46**	0.31	0.42	0.47	**0.50**	0.40
DL	0.66	0.58	0.50	**0.68**	0.44	0.39	0.33	**0.44**	0.43	0.36	0.38	**0.45**
General	0.72	0.76	0.78	0.74	0.55	0.65	0.62	0.57	0.44	0.44	0.46	0.46

aims to detect and eliminate the irrelevant words for recommendation systems, which poses a great challenge for current approaches to be robust when handling various user queries. Although a random deletion strategy reduces the overall performance on average, the positive improvement of deletion on some specific words indicates the potential benefits of noisy word deletion.

Implication 1: *Current query-based API recommendation approaches should be integrated with query reformulation techniques to be more effective.*

6.2 Data Sources for Query-based API Recommendation

In Section 4.1, we point out that current query-based API recommendation approaches face the problem of building a comprehensive knowledge base due to the lack of enough data such as query-API pairs. In Section 4.3, we further discover that there is a semantic gap between user queries and descriptions from the official documentation. Both the lack of enough data for knowledge base creation and the semantic gap increase the difficulty of accurate API recommendation based on only official documentation. Such challenges can not be easily solved by improving learning-based models or pattern-based models. One effective way to mitigate the difficulty is to involve Stack Overflow posts, as analyzed in Section 4.3. While Stack Overflow is only one type of data source, our analysis demonstrates that adding appropriate data sources can improve the performance of query-based API recommendation approaches.

Implication 2: *Apart from query reformulation, adding appropriate data sources provides another solution to bridge the gap between queries and APIs.*

6.3 Low Resource Setting in Query-based API Recommendation

In Section 4.1, we find that current learning-based methods do not necessarily outperform traditional retrieval-based methods. We attribute the results to the limited data such as query-API pairs in the query-based API recommendation task, which is a low-resource scenario [10], [25]. We also discover that pre-trained models such as BERT show superior performance in query reformulation in Section 4.2. This indicates that current pre-trained models can implicitly mitigate the semantic gap between user queries and official descriptions of APIs. Future work is suggested to explore how to make the best use of pre-trained models for query-based API recommendation based on limited available data.

Implication 3: *Few-shot learning with powerful pre-trained models can be a solution to further improve the performance of query-based API recommendation.*

6.4 User-defined APIs

In Section 5.2, we find that current code-based API recommendation approaches, no matter pattern-based or learning-based models, all face the challenge of recommending user-defined APIs. User-defined APIs have become the major bottleneck to further improve the performance of current code-based API recommendation approaches. However, as user-defined APIs usually do not appear in the training set, they can hardly be learned by machine learning methods or be mined by pattern-based methods. A possible solution used by current approaches [27], [33] is to regard the API as a code token and predict the token based on previous contexts. However, this solution also fails if the API token never appears in previous context. Thus, accurately predicting user-defined APIs should be one major direction of code-based API recommendation in future work.

Implication 4: *User-defined API recommendation is one major bottleneck for improving the performance of current code-based API recommendation approaches and remains unsolved.*

6.5 Query-based API Recommendation with Usage Patterns

In this paper, we only focus on testing whether an approach can recommend the correct APIs, but we believe developers can always benefit more from detailed information about

how to use the recommended APIs. A common method is to provide summaries such as the signature and constraints extracted from official documentation along with the recommended APIs. For example, KG-APISumm proposed by Liu *et al.* [38] provides a detailed summary of the recommended API class. However, official documentation sometimes cannot provide enough usage information about an API, which may cause API misuse. For instance, a fresh developer may search "*how to read a file*" in Python and the recommended API should be *fileObject.read()*, but without sufficient experience to use file operations, the developers may forget to close the file after reading it.

A possible solution to complement official documentation and avoid possible API misuse is to provide usage patterns from other developers. In the above example, a common usage pattern *open()*, *fileObject.read()*, *fileObject.close()* can prevent dangerous file operations. As there exist some pattern mining approaches on code, we can combine query-based API recommendation with code-based pattern mining methods for better providing the usage pattern.

Implication 5: *Code-based API recommendation approaches can provide usage patterns to enrich the results returned by query-based API recommendation approaches.*

6.6 Implications for Different Group of Software Practitioners

In this subsection, we conclude some implications for different group of software practitioners.

Software Researchers. For query-based API recommendation, we conclude that query reformulation techniques can bring significant improvement for current API recommendation approaches in Section 4.2. Despite of the effectiveness of query reformulation, it still remains unexplored on the factors that impact the performance of the technique. We believe a comprehensive study towards query reformulation can be an important future direction for API recommendation. For code-based API recommendation, we find that the major bottleneck for current approaches is user-defined API recommendation in Section 5.2. We suggest software researchers to focus more on the user-defined API recommendation for improving the practicability of API recommendation approaches.

Software Developers. As illustrated in Section 4.3, there exists a knowledge gap between official documentation and user queries, which limits the performance of current query-based API recommendation approaches. For developers who design new APIs, we believe adding more practical examples in the documentation or using more natural language descriptions would mitigate the knowledge gap. In Section 4.2.3, we find that current queries sometimes contain unnecessary information that confuse the API recommendation approaches. For developers who search for APIs, we believe that creating a query by using more professional words instead of unnecessary long descriptions can facilitate the search process.

7 Threat To Validity

In this section, we describe the possible threats we may face in this study and discuss how we mitigate them.

7.1 Internal Validity

Our research may face the following internal threats:

Baseline Re-implementation. In this paper, we re-implemented several baselines according to the code or replication packages released by their authors. However, as some baselines are not primarily designed for API recommendation, we slightly modified their code and adapted them into our task and our benchmark. For example, we limit the prediction scope of code completion baselines to only API tokens. Such adaptations may cause the performance of baselines to be slightly different from those in the original papers. To mitigate this threat and validate the correctness of our re-implementation, we refer to some related work that cites these baselines and confirm our experiment results with them.

Data Quality. We build APIBench-Q by manually selecting and labeling API-related queries from Stack Overflow and some tutorial websites. This process involved some human checks so that some subjective factors may influence the quality of our datasets. To mitigate this threat, we involve at least two persons to label one case and let one of our authors further check if the previous two persons give different opinions to the case. We also implement some rules to automatically filter out the cases that are explicitly unrelated to API recommendation.

Identification of User-defined APIs. We utilize commonly-used static import analysis to analyze the import statements in each source file and try to identify the implementation of imported libraries. To guarantee the quality of our benchmark dataset, we regard an API as a user-defined API only if we can find its implementation. However, since the completeness of static import analysis is still an open challenge, there may exist several user-defined APIs that cannot be identified. We will further refine the dataset when more advanced static important analysis tools are available.

7.2 External Validity

Our research may face the following external threats:

Data Selection. To the best of our knowledge, APIBench is the largest benchmark in API recommendation task. We try to make it more representative by selecting real-world code repositories from the most popular domains at GitHub and real-world queries from the largest Q&A forum StackOverflow according to several developer surveys [28], [66]. All findings in this empirical study are based on this dataset. However, there may still be slight differences when adapting our findings into other domains and datasets that we do not discuss in this paper.

Programming Language. Our study focuses on the API recommendation on Python and Java, and the findings included in this study may not be generalized to other programming languages. However, we believe the impacts of programming languages should not be significant as Python and Java are the most representative dynamically typed and statically typed languages, respectively.

8 Conclusion and Future Work

In this paper, we present an empirical study on the API recommendation task. We classify current work into query-based and code-based API recommendation, and build a benchmark named APIBench to align the performance of different recommendation approaches. We conclude some findings based on the empirical results of current approaches.

For query-based API recommendation approaches, we find that 1) recommending method-level APIs is still challenging; 2) query reformulation techniques have great potential to improve the quality of user queries thus they can help current approaches better recommend APIs. What's more, user queries also contain some meaningless and verbose words and even a simple word deletion method can improve the performance; 3) approaches built upon different data sources have quite different performances. Q&A forums such as Stack Overflow can greatly help mitigate the gap between user queries and API descriptions.

For code-based API recommendation, we emphasize the superior performance of current deep learning models such as Transformer. However, they still face the challenge of recommending user-defined APIs. We also find different contexts, such as different location of recommendation points and context length, can impact the performance of current approaches. Besides, current approaches suffer from recommending cross-domain APIs.

Based on the findings, we summarize some future directions on improving the performance of API recommendation. For query-based approaches, we encourage researchers to integrate query reformulation techniques with query-based API recommendation approaches to obtain better performance, but how to choose the best query reformulation strategy still remains as future work. We also believe some few-shot learning methods and different data sources can bridge the gap between user queries and knowledge base under low resource scenarios. For code-based approaches, we recommend future work to focus on improving the performance of user-defined API recommendation and train the approach on multiple domains instead of a single domain.

Apart from the findings and implications concluded in this paper, we also identify some future work that can be conducted for API recommendation. First, our paper focuses on benchmarking and provides an objective evaluation for all approaches. However, some approaches provide summaries for the recommended APIs that are not assessed in our study. For such approaches, subjective evaluation such as a developer survey can be conducted for verifying the quality of recommended API descriptions and usage information. Future work could consider complementing our work. Second, our empirical results show that query reformulation techniques are quite effective to improve the query quality. As this paper mainly focuses on API recommendation, we do not discuss different query reformulation techniques comprehensively. Future work can focus on studying the query reformulation techniques for facilitating downstream tasks.

We released our benchmark APIBench and all experiment results at Github[3]. We hope this empirical study can remove some barriers and motivate future research on API recommendation.

Acknowledgments

This research was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210920 of the General Research Fund). It was also supported by National Natural Science Foundation of China Grant under project No. 62002084, Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005), Stable support plan for colleges and universities in Shenzhen under project No. GXWD2020 1230155427003-20200730101839009, and the Major Key Project of PCL (Grant No. PCL2022A03, PCL2021A02, PCL2021A09).

References

[1] Geeks4geeks website. https://www.geeksforgeeks.org/, 2021.
[2] Java2s website. http://www.java2s.com/, 2021.
[3] Kode java website. https://kodejava.org/, 2021.
[4] Mohammad Masudur Rahman 0001 and Chanchal K. Roy. Improved query reformulation for concept location using coderank and document structures. *PeerJ PrePrints*, 5, 2017.
[5] AIDanial. Count lines of code. https://github.com/AlDanial/cloc, 2021.
[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.
[7] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.
[8] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. Automated query reformulation for efficient search based on query logs from stack overflow. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2021.
[9] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. The demo link for sequer. https://sequer-tpznovfjxa-uc.a.run.app/?query=, 2021.
[10] Mona Diab. Data paucity and low resource scenarios: Challenges and opportunities. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3612, New York, NY, USA, 2020. Association for Computing Machinery.
[11] Django. Django api reference. https://docs.djangoproject.com/en/3.2/ref/, 2021.
[12] Andrea Renika D'Souza, Di Yang, and Cristina V Lopes. Collective intelligence for smarter api recommendations in python. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 51–60. IEEE, 2016.
[13] Stack Exchange. Stack exchange data dump. https://archive.org/details/stackexchange, 2021.
[14] Flask. Flask api reference. https://flask.palletsprojects.com/en/2.0.x/api/, 2021.
[15] J. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76:378–382, 1971.
[16] Apache Software Foundation. Lucene. https://lucene.apache.org/, 2021.
[17] Eclipse Foundation. The eclipse ide. https://www.eclipse.org/eclipseide/, 2021.
[18] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.

3. The address of benchmark is at https://github.com/JohnnyPeng18/APIBench

[19] Jaroslav M. Fowkes and Charles Sutton. Parameter-free probabilistic API mining across github. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 254–265. ACM, 2016.

[20] Google. Android api reference. https://developer.android.com/reference, 2021.

[21] Google. The google news word2vec model. https://code.google.com/archive/p/word2vec/, 2021.

[22] Google. The interface of google prediction service. http://suggestqueries.google.com/complete/search?, 2021.

[23] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[24] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. Pyart: Python api recommendation in real-time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1634–1645. IEEE, 2021.

[25] Michael A. Hedderich, Lukas Lange, Heike Adel, Jannik Strötgen, and Dietrich Klakow. A survey on recent approaches for natural language processing in low-resource scenarios, 2021.

[26] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847. IEEE Computer Society, 2012.

[27] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 293–304. IEEE, 2018.

[28] Jetbrains. Python developer survey conducted by jetbrains and python software foundation. https://www.jetbrains.com/lp/python-developers-survey-2020/, 2020.

[29] JetBrains. The intellij idea ide. https://www.jetbrains.com/idea/, 2021.

[30] JetBrains. The pycharm ide. https://www.jetbrains.com/pycharm/, 2021.

[31] Elise Jing, Kristiana Schneck, Dennis Egan, and Scott A. Waterman. Identifying introductions in podcast episodes from automatically generated transcripts, 2021.

[32] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[33] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. The replication package of travtrans. https://github.com/facebookresearch/code-prediction-transformer, 2021.

[34] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. *ACM Comput. Surv.*, 54(8), oct 2021.

[35] lanwuwei. A pre-trained bert on stackoverflow corpus. https://github.com/lanwuwei/BERTOverflow, 2021.

[36] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. ICSE '19, page 795–806. IEEE Press, 2019.

[37] Bohan Li, Yutai Hou, and Wanxiang Che. Data augmentation approaches in natural language processing: A survey. *AI Open*, 2022.

[38] Mingwei Liu, Xin Peng, Andrian Ma rcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. Generating query-specific class api summaries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 120–130, 2019.

[39] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 545–549. IEEE Computer Society, 2015.

[40] Edward Ma. Nlp augmentation. https://github.com/makcedward/nlpaug, 2019.

[41] Matplotlib. Matplotlib api reference. https://matplotlib.org/stable/api/index.html, 2021.

[42] Microsoft. Github website. https://github.com/, 2021.

[43] Microsoft. The visual studio code editor. https://code.visualstudio.com/, 2021.

[44] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[45] Benjamin Newman, Prafulla Kumar Choubey, and Nazneen Rajani. P-adapters: Robustly extracting factual information from language models with diverse prompts, 2021.

[46] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, 2016.

[47] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 858–868. IEEE, 2015.

[48] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 69–79. IEEE, 2012.

[49] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. Focus: A recommender system for mining api function calls and usage patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1050–1060. IEEE, 2019.

[50] Numpy. Numpy api reference. https://numpy.org/doc/stable/reference/, 2021.

[51] Oracle. Java se 8 api documentation. www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html, 2021.

[52] Pandas. Pandas api reference. https://pandas.pydata.org/docs/reference/index.html, 2021.

[53] Python. Python standard library. https://docs.python.org/3/library/, 2021.

[54] Maithra Raghu and Eric Schmidt. A survey of deep learning for scientific discovery, 2020.

[55] Mohammad Masudur Rahman and Chanchal Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. 06 2018.

[56] Mohammad Masudur Rahman and Chanchal Roy. Nlp2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 714–714, 2018.

[57] Mohammad Masudur Rahman and Chanchal Roy. The replication package for nlp2api. https://github.com/masud-technope/NLP2API-Replication-Package, 2021.

[58] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359. IEEE, 2016.

[59] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 731–747, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[61] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[62] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[63] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[64] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[65] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and