API Recommendation for Machine Learning Libraries: How Far Are We?

Moshi Wei
York University
Toronto, Canada
moshiwei@yorku.ca

Yuchao Huang
Institute of Software Chinese Academy of Sciences
yuchao2019@iscas.ac.cn

Junjie Wang
Institute of Software Chinese Academy of Sciences
junjie@iscas.ac.cn

Jiho Shin
York University
Toronto, Canada
jihoshin@yorku.ca

Nima Shiri Harzevili
York University
Toronto, Canada
nshiri@yorku.ca

Song Wang
York University
Toronto, Canada
wangsong@yorku.ca

ABSTRACT

Application Programming Interfaces (APIs) are designed to help developers build software more effectively. Recommending the right APIs for specific tasks is gaining increasing attention among researchers and developers. However, most of the existing approaches are mainly evaluated for general programming tasks using statically typed programming languages such as Java. Little is known about their practical effectiveness and usefulness for machine learning (ML) programming tasks with dynamically typed programming languages such as Python, whose paradigms are fundamentally different from general programming tasks. This is of great value considering the increasing popularity of ML and the large number of new questions appearing on question answering websites.

In this work, we set out to investigate the effectiveness of existing API recommendation approaches for Python-based ML programming tasks from Stack Overflow (SO). Specifically, we conducted an empirical study of six widely-used Python-based ML libraries using two state-of-the-art API recommendation approaches, i.e., BIKER and DeepAPI. We found that the existing approaches perform poorly for two main reasons: (1) Python-based ML tasks often require significant long API sequences; and (2) there are common API usage patterns in Python-based ML programming tasks that existing approaches cannot handle. Inspired by our findings, we proposed a simple but effective frequent itemset mining-based approach, i.e., FIMAX, to boost API recommendation approaches, i.e., enhance existing API recommendation approaches for Python-based ML programming tasks by leveraging the common API usage information from SO questions. Our evaluation shows that FIMAX improves existing state-of-the-art API recommendation approaches by up to 54.3% and 57.4% in MRR and MAP, respectively. Our user study with 14 developers further demonstrates the practicality of FIMAX for API recommendation.

CCS CONCEPTS

• Information systems → Recommender systems; **• Computing methodologies → Machine learning**.

KEYWORDS

API recommendation, Python-based machine learning library, empirical software engineering

ACM Reference Format:
Moshi Wei, Yuchao Huang, Junjie Wang, Jiho Shin, Nima Shiri Harzevili, and Song Wang. 2022. API Recommendation for Machine Learning Libraries: How Far Are We?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549124

1 INTRODUCTION

Application Programming Interfaces (APIs) are built-in functions in software libraries that help developers build software more effectively. As machine learning recently made great progress in both theory and application, there is increasing interest in developing machine learning applications. While there are many publicly available open-source machine learning libraries and APIs, searching the right APIs for specific tasks is not easy, especially for the plenty of green hands in the field of machine learning [50].

Although many API recommendation approaches that can help retrieve APIs with high accuracy have been proposed and extensively studied [16, 17, 19, 35], most existing approaches have been evaluated mainly on general programming tasks using programming languages such as Java. Little is known about their practical effectiveness and usefulness for ML programming tasks with dynamically typed programming languages such as Python. Apart from the different nature of programming languages, machine learning application development has different paradigms compare to traditional application development (relatively more deterministic and less statistically-orientated) [7, 50], which is statistically-orientated and requires many algorithms, mathematical operations, and data operations [12, 29, 43]. In this paper, we investigate the effectiveness of existing API recommendation approaches for Python-based ML programming tasks, which is of great value considering the increasing popularity of machine learning practices and the overwhelming number of public asked machine learning questions in question answering websites such as Stack Overflow (SO). For example, there are more than 11K questions on Stack Overflow about API recommendation for TensorFlow (an open-source library for machine learning) with about 300 new questions emerging each week[1]. Our study focuses on the following research questions:

RQ1: What is the performance of existing API recommendation approaches on Python-based ML programming tasks?

To answer this question, we present an empirical study to explore the performance of existing state-of-the-art API recommendation approaches, i.e., BIKER [19] and DeepAPI [17], on Python-based ML programming tasks.

Specifically, first of all, we collect 80K Python-based ML programming questions related to six popular machine learning libraries from SO. Then we retrain BIKER [19] and DeepAPI [17] with our Python-based ML question dataset for fair evaluation and we evaluate their performance using 1K randomly selected ML questions (excluded from the training data). Our experiment results show that there exists significant performance decline of these approaches on Python-based ML questions. We then perform an in-depth analysis to explore the reasons. Our analysis reveals that there exist two major reasons. First, most traditional Java programming tasks require only one API to solve, while ML questions often require many more APIs because ML development tasks often require customization of data processing, feature engineering, model architecture, optimization function, and hyperparameters, etc. Specifically, the average length of API sequence in the answers to Java programming tasks is 1.42, while the average length is 5.50 for ML tasks. Second, most existing approaches only focus on increasing the hit rate of the first correct API recommended while ignoring the completeness of the answers recommended. Moreover, the multiple APIs of Python-based ML tasks pose greater challenges to the recommendation tasks for Python-based ML tasks. Note that, we also observe several other reasons for the performance decline, which are closely related to the nature of machine learning (details are in Section 7.1).

In addition, we have also observed that there exist common library-specific API usages that can be useful for further improving the API recommendation for Python-based ML programming tasks. For example, when constructing a machine learning model in Keras, *Keras.model.Sequential* has to be used as a container for model layers such as *keras.layers.Conv2D, keras.layers.Dense.* Current API recommendation approaches cannot capture the above information about API usage as they do not consider the relationship between API calls when recommending APIs for a programming task.

RQ2: Can we improve the performance of the existing API recommendation approaches on Python-based ML tasks?

Based on our findings from RQ1, we propose a simple but effective booster, i.e., FIMAX, which can significantly improve the performance of existing API recommendation approaches for Python-based ML programming tasks by leveraging API usage patterns of ML libraries.

Specifically, we propose to use the frequent itemset mining technique [26] to identify API usage patterns from the API call sequences in the answers of ML programming tasks posted in SO. We then extend the recommendation results of the existing approaches with the API usage patterns. Our evaluation shows that the proposed approach improves the existing state-of-the-art API recommendation approaches by up to 54.3% and 57.4% in MRR and MAP, respectively. We have also conducted a user study in which 14 developers are divided into two groups using different tools to answer 20 ML questions randomly sampled from the testing dataset. On average, the group using the FIMAX can improve answer correctness by 36% and save answering time by 40%.

This paper makes the following contributions:

- We perform the first empirical analysis of existing API recommendation approaches on Python-based ML programming tasks, reveal their performance degradation on Python-based ML programming tasks, and summarize the reasons for the degradation.
- We propose a simple but effective approach, i.e., FIMAX, to augment the API sequences retrieved by existing approaches to boost their performance of API recommendations.
- Both our quantitative evaluation and user study show that FIMAX can help developers find the correct APIs for Python-based ML programming tasks more efficiently and accurately, compared with state-of-the-art baselines.
- We release the source code of our tool and the dataset of our experiments to help other researchers replicate and extend our study[2].

This paper is organized as follows. Section 2 describes the background of the API recommendation research field. Section 3 presents the setup of our empirical study. Section 4 shows the details of our experiment and the results. Section 5 presents the performance of our proposed approach. Section 6 shows our case study. Section 7 discusses open questions about this study and threats to the validity of our work. Section 8 presents the related work. Section 9 is the conclusion of the paper.

2 BACKGROUND ON API RECOMMENDATION

There are many existing approaches for API recommendation [6, 8–11, 23, 25, 34, 38, 49]. These approaches can be divided into two orthogonal categories, i.e., information retrieval-based approach and deep-learning-based approach. In this paper, two modern approaches, one for each category, are studied.

2.1 Information Retrieval based API Recommendation

Information retrieval (IR) based API recommendation approaches [10, 19, 25, 34], as the name suggests, leverage information retrieval techniques for recommending APIs for a given programming task described in natural language. These approaches first apply the source code parsing algorithms and heuristics to extract API call sequences from the answers of related SO questions and tokenize them into bag-of-words. Next, these approaches create indexes for features extracted from the collected SO questions. Given a query,

[1]https://stackoverflow.com/questions/tagged/tensorflow

[2]https://doi.org/10.5281/zenodo.6360250

the approaches preprocess the query and then compute the similarities between the query and the collected questions using the built indices. The most similar matches are returned as output.

The state-of-the-art information retrieval-based approach for API recommendations is BIKER [19]. It uses both Stack Overflow questions and API documentation to recommend APIs for Java programming tasks. BIKER considers the API recommendation problem as a two-step task. Given an input query, BIKER first retrieves the k most similar questions by using the text similarity between the query and all questions in its knowledge base. Then it creates a list of API candidates from the top k questions and re-ranks the APIs according to the text similarity between the query and the documentation of an API. In this paper, we select BIKER as our baseline to represent the state-of-the-art IR-based API recommendation technique.

2.2 Deep Learning based API Recommendation

Most deep learning-based API recommendation approaches treat the API recommendation as a translation task and apply an end-to-end architecture where the deep learning model takes a query sentence as input and returns a list of recommended APIs as output. Most deep learning based API recommendation approaches train the models on the training data (i.e., a large corpus of question and answer pairs) and use the trained models for inference. In contrast to the token similarity ranking strategy of IR-based API recommendation approaches, deep learning-based API recommendation approaches focus on learning the semantic connection between queries and the API answers. The deep learning model learns the semantics of a query from the training data and recommends APIs based on the deep semantic connection knowledge it learns between queries and the API answers.

The state-of-the-art deep learning-based API recommendation approach is DeepAPI [17]. The core model of DeepAPI is a Recurrent Neural Network(RNN) based Encoder-Decoder model, which is originally used for neural machine translation(NMT). DeepAPI was trained on a large corpus of question-API pairs extracted from GitHub repositories and treats the API recommendation problem as a sequence generation task where the input is a programming query and the output is an API sequence. In this paper, we use DeepAPI as our baseline to represent the state-of-the-art deep learning-based API recommendation.

3 EMPIRICAL STUDY SETUP

This section describes the research questions of our work, data collection approach, and analysis methodology.

3.1 Research Questions

Our work is organized by the following two research questions:

RQ1: What is the performance of existing API recommendation approaches on Python-based ML programming tasks?

In this RQ, we set out to investigate whether the state-of-the-art existing API recommendation approaches, i.e., BIKER and DeepAPI, perform well on Python-based ML questions.

RQ2: Can we improve the performance of the existing API recommendation approaches on Python-based ML programming tasks?

This question aims to explore possible solutions that can improve existing API recommendation approaches based on our findings from RQ1.

3.2 Subjects of Study

To investigate the performance of existing API recommendation approaches on Python-based ML questions, we select six popular Python ML libraries according to the number of downloads in the Python Package Index (PyPI) [2], which is the Python library management program.

We manually identify the machine learning-related packages from the top PyPI package list[3] , which is ranked by number of downloads. According to PyPI statistics [1] (as of Dec. 2021), the six libraries that rank by monthly downloads are NumPy, Pandas, Scikit-Learn, PySpark, TensorFlow, and Keras. NumPy [31] is a fundamental library used in Python ML development mainly for numeric array operations. Pandas [24] is a library for data analysis of tabular data such as data tables. It provides tabular display of parallel data and also APIs covering basic table operations such as filtering, sorting, and indexing. Scikit-learn [32] is an important library in statistical machine learning library that provides machine learning and scientific algorithms such as matrix operations, algebraic equations, and differential equations [41]. TensorFlow [13] and Keras [21] are two popular deep learning libraries. TensorFlow, developed by Google, is one of the most commonly used machine learning libraries in the industry as it supports many industry-friendly features such as distributed training and a visual debugging environment [50]. Keras offers a user-friendly stack-style API for building and training deep learning models. It encapsulates the detailed execution process in high-level APIs. PySpark [14] is a Python adapter of Apache Spark. It allows developers to use Spark application features such as Spark SQL, Distributed Data Frame, etc. in Python.

The libraries studied cover most of the current industrial practice of machine learning and also represent the key aspects of machine learning developments. In addition, they cover the area of data preprocessing and data object processing (NumPy and Pandas), scientific computing (SciPy and Scikit-learn), distributed machine learning (PySpark), and deep learning (TensorFlow and Keras). Although NumPy and Pandas are not libraries for building neural network structures, they are integral to machine learning development because they are widely used as the basis for data object computation in machine learning model development. NumPy APIs are typically utilised in the middle of the TensorFlow modelling API sequences. Removing the NumPy APIs would leave the API sequences incomplete. Thus, in this work we also experiment with Numpy and Pandas.

Table 1 shows the details of the six libraries studied. The number of APIs is calculated by counting the number of documented function nodes in the latest version of the packages using a Python AST parser. After removing duplicates for each package, we get the number of unique APIs in our experiment dataset. The number

[3]https://hugovk.github.io/top-pypi-packages/

Table 1: The details of studied machine learning libraries in this work

Library	Description	#APIs in documents	# Unique APIs in dataset	#SO questions
Numpy	A library for multi-dimensional arrays and matrices operations.	1,373	1,151	29,771
Pandas	A library for data manipulation and analysis.	1,367	913	16,197
Scikit-learn	A library for machine learning algorithms.	1,154	1,082	6,376
Keras	An interface for artificial neural networks.	1,407	1,233	8,728
TensorFlow	A library for deep learning developed by Google.	8,028	1,591	10,263
PySpark	An interface for Apache Spark in Python.	860	785	8,861

Table 2: Performance of BIKER and DeepAPI on Python-based ML questions

Dataset	Approach	Evaluation Metrics	
		MRR	MAP
Python ML	BIKER	0.271	0.115
	DeepAPI	0.176	0.068
Java JDK	BIKER	0.554	0.505
	DeepAPI	0.188	0.153

of SO posters for each package is collected from our Python ML dataset, as shown in Section 3.3.

3.3 SO Post Collection

To create our Python-based ML question dataset, we follow existing work [19] to extract programming questions and their accepted answers from Stack Overflow using the data explorer provided by Stack Exchange [3]. We select questions with the six library names as Stack Overflow question tags to retrieve questions and only questions with accepted answers remain.

APIs are extracted from the accepted answer using heuristics. Specifically, we developed a heuristic parser to extract the method names from the code snippet and the package and class names are inferred based on the package import information. If a question contains multiple APIs, we concatenate all APIs into a list of APIs to answer the question. We only select answers that contain at least one API and all APIs are in the same library to reduce noise in the training dataset. As a result, we collected a total of 80,196 question-API pairs for these libraries.

3.4 Evaluation Metrics

We evaluate API recommendation approaches using Mean Reciprocal Rank (MRR) and Mean Average Rank (MAP), which are two commonly used evaluation metrics in information retrieval and recommendation system evaluation [19, 22, 35, 36, 44, 46–48]. Both MRR and MAP range from 0 to 1. MRR describes the rank of the first correctly recommended API in the recommendation list. It is calculated by the inverse rank of the first match [19]. A high MRR indicates that the rank of the first correct match is high. MAP checks the ranks of all correct matches instead of focusing only on the first correct answer. A high MAP indicates that there are multiple correct matches in the recommendation and thus the recommendation is complete.

4 RQ1: PERFORMANCE OF EXISTING APPROACHES ON PYTHON-BASED ML QUESTIONS

To investigate RQ1, we train and evaluate BIKER and DeepAPI on our Python-based ML questions collected from Stack Overflow (see Section 3.3 for details).

In addition, an in-depth analysis was conducted to investigate possible reasons for the difference in performance of the two approaches on traditional programming tasks and Python-based ML programming tasks.

4.1 Experimental Setup

Training: Since BIKER and DeepAPI were designed for Java JDK API recommendation, to make it applicable for Python-based ML questions, we replace the Java question data with the Python-based ML question data for retraining the models. Note that, BIKER uses both question titles and API documents for recommendation, we also replace the Java documentation with the Python documentation.

To collect Python documents, we use the Python AST parser to collect the docstrings of each Python library used in this work. For each library, we run the AST parser from the root and collect all docstrings in the "FuctionDef" node and the "ClassDef" node. To evaluate DeepAPI on Python-based ML questions, we train and tune the DeepAPI model from scratch on our Python-based ML question dataset with the same configuration as reported in its original paper [17] on a single Nvidia V100 GPU with 32GB memory.

Testing: To evaluate the performance of the examined two API recommendation models, we randomly select 1k samples for each library for building the test datasets. According to previous studies on Stack Overflow [28, 40, 42], Stack Overflow contains not only API recommendation questions, but also other types of questions, such as comparing different implementations, asking for an explanation, and asking for program debugging, etc. Therefore, to reduce noises in the test data, we manually examine each question to remove the questions that are not suitable for API recommendations. The questions removed in our manual analysis either do not ask for API recommendations or provide information that is too general to derive an API recommendation from. For example, questions that ask for explanations, such as "*Why am I not seeing NumPy's Deprecation Warning?*", questions that ask for debugging help, such as "*MNIST ValueError when checking target in Keras*", questions that are too general to recommend APIs, such as "*Constraints not working in Optimization using SciPy*". The authors of this work independently go through each question and identify the API recommendation-related questions. The agreement among the researchers measured by Cohen's Kappa coefficient is 0.87, which is a relatively high-level of agreement. We remove the irrelevant questions and select the first 100 samples for each library from the remaining samples as the final test set.

Note that, we have removed all test data from the training data for the fair evaluation.

4.2 Performance Difference

Table 2 shows the MRR and MAP of BIKER and DeepAPI on the Python ML question dataset. In addition, we also showed the performance of BIKER and DeepAPI on the Java JDK question dataset used in their original papers [17, 19].

In general, the performance of both approaches significantly (p-value <0.01) decrease when applied to the Python ML dataset compared to their performance on the Java JDK dataset.

For BIKER, the MRR declines from 0.554 on the Java JDK dataset to 0.271 on the Python ML dataset (declines 51.0%), and the MAP declines from 0.505 to 0.115 (declines 77.2%). For DeepAPI, we can observe a similar trend for both MRR and MAP, i.e., MRR declines by 6.0% and MAP declines by 55.0%.

We also note that for the Java JDK dataset, the MRR and MAP of both approaches are at a similar level, while MAP for the Python ML dataset is about half the MRR for both approaches. Since MAP measures the ranks of all correct matches instead of focusing only on the first correct answer like MRR, the above results show that the existing API recommendation approaches have challenges on the API recommendation completeness, i.e., they can hardly capture the entire set of correct answers.

4.3 Analysis of Performance Decline

Motivated by the significant performance difference showed in Section 4.2, we further investigated the two datasets (i.e., Java JDK question dataset and Python-based ML question dataset) and the recommended APIs of both BIKER and DeepAPI. We found two main reasons contributing to the performance decline of BIKER and DeepAPI on the Python-based ML question dataset, i.e., compared to the Java question data, answers of the Python ML questions often require more APIs (see Section 4.3.1 for details) and the existing API recommendation approaches mainly focus on increasing the hit rate of the first correct API recommended while ignoring the completeness of the answers recommended and do not consider the common API usages (see Section 4.3.2 for details). Note that in addition to these two major reasons, we also observed some other minor reasons that could decline the performance of the existing API recommendation approaches. The details can be found in Section 7.1.

4.3.1 ***Python ML Questions Requires More APIs.*** We find that the average length of API call sequence in the answers of the Python-based ML questions is longer than that of the answers for Java SDK questions, which makes the recommendation for Python-based ML question more challenging. Specifically, the average length of the API call sequence in the ground-truth answers of the Java JDK questions is 1.42, while the average length of the API call sequence in the answers for Python-based ML questions is 5.50 In other words, one API can solve a Java JDK question, while five APIs are required to solve a Python-based ML question on average. The median API call sequence length also supports this observation, i.e., one in Java versus five in Python ML.

Due to the fact that most answers to Java JDK questions only contain a single API, existing API recommendation approaches such as BIKER and DeepAPI mainly focus on increasing the probability of the first correct answer (i.e., MRR), while ignoring the completeness of the recommended APIs (i.e., MAP). Specifically, given a question, BIKER first obtains candidate APIs from a list of similar questions, and then re-rank the collected APIs according to the similarity between the question title and the documentation of each API to ensure the correctness of the first answer without considering other APIs that have low similarity scores to the question title if even they are essential for answering the question. This hurts the completeness of the recommendation. For DeepAPI, it uses an LSTM encoder-decoder sequential model designed for the sequential task. This mitigates the problem BIKER faces yet still suffers from low performance. Considering that Python-based ML questions usually contain 5 or more APIs, the API recommendation in this scenario should consider both MRR and MAP, i.e., the first correct answer and all correct answers are ranked higher.

4.3.2 ***Existing Approaches Did not Consider Common APIs Usages.*** Table 3 shows an Python-based ML question, its answer, and the recommended APIs by using BIKER. The question is asking

Table 3: An example of Python-based ML question (Bold APIs are APIs that are matched by BIKER, underlined APIs are APIs that are missed by BIKER).

Question: convert vgg16 shape output from 4096 features to 2048

Accepted code snippet to answer this question:

vgg16_model = keras.applications.vgg16.**VGG16()**
model = Sequential()
for layer in vgg16_model.layers[:-1]:
    model.**add**(layer)
model.layers.pop()
# Freeze the layers
for layer in model.layers:
    layer.trainable = False
# Add 'softmax' instead of earlier 'prediction' layer.
model.**add**(Dense(2048, activation='softmax'))
# Check the summary, and yes new layer has been added.
model.summary()

Ground-truth APIs:
keras.VGG16, keras.Sequential, **keras.add**, keras.pop, keras.Dense, keras.summary

Recommended APIs by BIKER:
keras.reshape, keras.concatenate, keras.shape, **keras.add**, keras.Input, keras.ones, keras.pad_sequences, **keras.VGG16**, keras.transpose, keras.arange

Table 4: Occurrence of API usage patterns

Base	Add	co-occurrence	confidence
keras.add	keras.Dense	972	0.7086
keras.add	keras.Sequential	1,188	0.6103

for the conversion of 4096 output shape to 2048 output shape. In the table, we also show the accepted code snippet to answer this question from which we can see that the answer of API sequence includes six APIs. In the code snippet, we bold the APIs that are found by BIKER and underline the APIs that are not found by BIKER.

Specifically, it first loads the VGG16 model except for the last layer and freezes the model layers. Then, it adds a new dense layer to the model with the required feature size. BIKER can correctly suggest "*keras.add*" and "*keras.VGG16*", but miss the prerequisite API, "*keras.Sequential*". In Keras, "*keras.Sequential*" acts as a container for all model layers, which means that the layer operation APIs must be called after a model container is initialized with "*keras.Sequential*". BIKER fails to recognize this API usage pattern. We also observe similar cases in DeepAPI's recommendation results.

The above example shows that existing API recommendation approaches often neglect the relationships between the APIs involved when recommending APIs for a given question. Motivated by this, we further investigate the co-occurrence of the involved APIs in our experimental dataset. The results are in Table 4, from which we can see that the API "*Keras.add*" occurs 1947 times, and "*Keras.Sequential*" co-occurs 1188 times together with "*Keras.add*" out of the 1947 times (i.e., 61%). This reveals the frequent usage patterns of APIs in solving a particular task, i.e., two or more APIs co-occur frequently. We have also observed there exist numerous pairs of APIs that are highly likely to co-occur, which can be considered as API usage patterns.

The above analysis motivates us to extend the API recommendation result by existing tools (e.g., BIKER and DeepAPI) by including API co-occurrence relationships to improve the performance of API recommendation. Specifically, when an API is recommended, our approach can identify potential API usage patterns related to it and further append the involved APIs from the API usage patterns into the recommended API list.

We believe that such an extension could improve current API recommendation approaches by increase the completeness of the APIs recommended.

5 RQ2: SOLUTION TO ENHANCE EXISTING API RECOMMENDATION APPROACHES

RQ1 has demonstrated the significant performance degradation of existing API recommendation approaches on Python-based ML questions. Inspired by the findings of RQ1, we propose a frequent itemset mining based approach, named FIMAX, that aims to improve the performance of existing API recommendation approaches for Python-based ML questions by extending existing API recommendation results with API usage information, i.e., co-occurrence relationships of APIs.

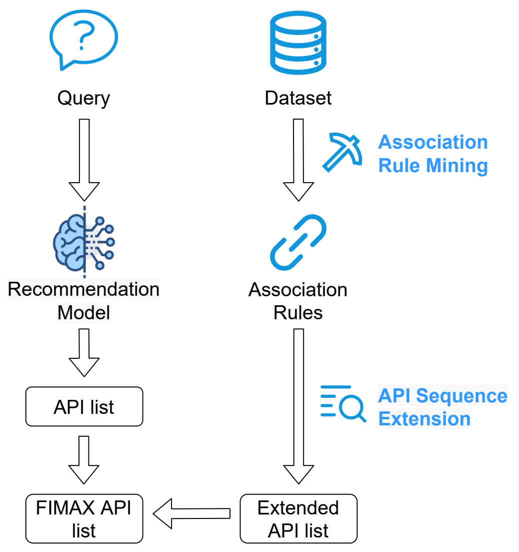


Figure 1: Overview of FIMAX

5.1 Frequent Itemset Mining for API Extension

Figure 1 shows the workflow of FIMAX . First, FIMAX generates the co-occurrence relation-based API usage patterns of APIs with association rule mining technique on API call sequences from the answers of ML programming tasks. Then, FIMAX further extends the recommendation from an API recommendation model with the mined the API usage patterns.

5.1.1 ***Association Rule Mining.*** FIMAX first applies the Apriori algorithm [5] to create a set of API association rules (i.e., API usage patterns) from the Python-based ML question dataset. Specifically, we collect the API sequence from the answer of each SO post in the Python-based ML question dataset, which serves as input to the Apriori algorithm for rule mining. The Apriori algorithm generates a pattern hypothesis by randomly selecting two or more APIs as the base itemset A and then randomly selecting one or more APIs as the extension itemset B. The algorithm calculates the confidence and support of the pattern hypothesis $\{A \Rightarrow B\}$ and compares it to the confidence and support threshold specified by the user. The Apriori algorithm accepts a association rule if the *confidence* and *support* of the rule are higher than the threshold values specified [5].

Specifically, *Support* indicates the frequency of an API association rule with respect to the entire dataset. *Confidence* indicates the percentage of one or more extended API(s) (e.g., item set B) found to be true given a base rule (e.g., item set A).

Let A, B be the antecedent and the consequent mined from a list of API T, the *support* of $A \Rightarrow B$ over T is

$$support(A \Rightarrow B) = \frac{freq.(A, B)}{|T|}$$

and the *confidence* of $A \Rightarrow B$ is

$$confidence(A \Rightarrow B) = \frac{freq.(A, B)}{freq.(A)}$$

As an example, the Apriori algorithm proposes an API usage pattern {*"tensorflow.sessi-on"*$\Longrightarrow$*"tensorflow.run"*}. Then the algorithm

Table 5: An example of API extension of FIMAX

APIs recommended:
numpy.range, numpy.reshape, numpy.arange

Rule A:
$\{numpy.range \Rightarrow numpy.empty, support = 0.0103\}$
Rule B:
$\{numpy.range \Rightarrow numpy.append, support = 0.0168\}$
Rule C:
$\{numpy.reshape, numpy.arange \Rightarrow numpy.array, support = 0.02204\}$

Extended API list:
numpy.range, numpy.reshape, **numpy.append**, **numpy.array**, numpy.arange

calculates the *confidence* and *support* for this pattern. The *confidence* can be interpreted as the percentage of occurrences of *"tensorflow.run"* given *"tensorflow.session"*, and the *support* of {*"tensorflow.session"*⟹*"tensorflow.run"*} is the percentage of the co-occurrences of *"tensorflow.session"* and *"tensorflow.run"* over the size of the entire corpus. If both *confidence* and *support* are greater than the specified thresholds, the proposed rule{*"tensorflow.session"*⟹*"tensorflow.run"*} will be added to the table of item set patterns along with the *confidence* and *support* values.

5.1.2 ***API Sequence Extension.*** Given a list of recommended APIs R, generated by an API recommendation approach, e.g., BIKER, FIMAX searches for the extension rules for each API in the list. If there are rules that match a particular API, FIMAX appends the associated APIs to R according to the association rules created in Section 5.1.1 for that API. If there are multiple rules that are eligible for extension, only the rule with the highest support score will be utilized. The final API recommendation list consists of the top K original APIs concatenated with the extended APIs. The extended APIs are ordered by confidence score.

Table 5 shows an example of the extension. Following the extension algorithm, FIMAX uses Rule B and Rule C for extension. In case there are duplicate APIs after extension. For a repeated recommendation, we keep the first occurrence and remove the duplicates.

5.2 Experiment Setup

As we discussed in Section 5.1, FIMAX applies Apriori algorithm for association rule mining, which has two parameters, i.e., *support* and *confidence*, that can significantly affect its output. To find the best values for these two parameters, we tune them together and experiment with support threshold values from 0.002 to 0.01 with a step of 0.001, and confidence thresholds with values from 0.1 to 0.9 with a step of 0.1. Please note that both BIKER and DeepAPI can recommend many APIs for a given question, e.g., BIKER recommends up to 50 APIs, as a result there will be a large number of candidate rules if all the recommended APIs are considered, while only a few API candidates are likely to be correct in practice. Extending the wrong API would degrade the performance of a recommendation approach. Therefore, we limit the number of APIs to be extended to less than a threshold, i.e., K.

For our tuning, we perform a grid search with all the above combinations of support threshold, confidence threshold, and top K and calculate the MRR value of each combination for performance comparison. The result shows that the BIKER+FIMAX model performs best when the confidence threshold is equal to 0.1, the support threshold is equal to 0.02 and top_k is equal to 6, while the DeepAPI+FIMAX model performs best when the confidence threshold is equal to 0.1, the support threshold is equal to 0.02, and top_k is equal to 10. The reason for the different best parameter settings for BIKER and DeepAPI can be these two approaches adopt different mechanisms in API recommendation and generate different recommended APIs. It also shows the necessity of parameter tuning when applying FIMAX to a new API recommendation approach. We use a Nvidia v100 GPU to train models and recommend APIs for DeepAPI, and we use Intel i7-4790 CPU, to recommend APIs for BIKER and FIMAX.

5.3 Performance Analysis of FIMAX

We evaluate the effectiveness of FIMAX on boosting two typical API recommendation approaches, i.e., BIKER and DeepAPI, on Python-based ML questions by comparing the performance of original BIKER/DeepAPI to the performance of these two approaches armed with FIMAX.

Table 6 shows the evaluation results for BIKER, DeepAPI, BIKER+FIMAX and DeepAPI+FIMAX on questions from each Python ML library. From the table, we can see that the FIMAX significantly increased the performance of both models. Overall, the MRR and MAP of BIKER increased by 48.36% and 53.46%, respectively, after applying the FIMAX. The MRR and MAP of DeepAPI increase by 54.28% and 57.36%, respectively, after applying FIMAX. Moreover, the performance increase of MAP is larger than MRR for both models, suggesting that FIMAX has a positive effect on the completeness of API recommendations. Our Wilcoxon signed-rank test ($p<0.05$) also suggests that BIKER and DeepAPI can achieve significantly better performance after applying FIMAX.

In addition, we can also see that BIKER+FIMAX has a noticeable improvement over the performance of BIKER for each library. The MAP of BIKER in the PySpark package shows the most significant improvement, i.e., a 126.46% increase after applying FIMAX. The main reason for this improvement might be that PySpark is an analytics engine for Big Data and most of its API sequence follows the Map-Reduce related patterns that can be learned by FIMAX.

For DeepAPI, we see that the MRR for the NumPy library increased by 127.6% and the MAP increased by 142.7% after the FIMAXwas applied, which is the most significant improvement of DeepAPI after applying FIMAX. One of the possible reasons for this improvement is that NumPy is designed for manipulating arrays and matrices exclusively. Thus, its API usage patterns can be much more targeted than other libraries, which makes them easy for FIMAX to learn.

Note that although FIMAX can significantly improve BIKER and DeepAPI on most Python ML libraries regarding both MRR and MAP, we observe a performance decline on the Scikit-learn library of DeepAPI after applying FIMAX. Specifically, the MAP of

Table 6: Performance of FIMAX on BIKER and DeepAPI.

Library	BIKER		BIKER+FIMAX		Improvement		DeepAPI		DeepAPI+FIMAX		Improvement	
	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP
Numpy	0.4369	0.1639	0.5689	0.2114	30.75%	27.39%	0.2160	0.0699	0.4918	0.1697	127.6%	142.7%
Pandas	0.1131	0.0514	0.1955	0.0635	67.50%	21.48%	0.2286	0.0791	0.2749	0.0975	20.22%	23.24%
Scikit-learn	0.0187	0.0037	0.0417	0.0417	100.5%	96.95%	0.1751	0.0656	0.1888	0.0604	7.786%	-7.99%
Keras	0.0447	0.0098	0.0612	0.0237	7.021%	88.95%	0.2124	0.0765	0.2662	0.0986	25.34%	28.85%
Tensorflow	0.2543	0.0666	0.3578	0.1137	37.40%	61.39%	0.2954	0.1020	0.4060	0.1497	37.41%	46.73%
Pyspark	0.2294	0.0422	0.3998	0.0969	71.31%	126.7%	0.2513	0.0838	0.4578	0.1413	82.19%	68.49%
Overall	0.1829	0.0563	0.2714	0.0864	48.36%	53.46%	0.2298	0.0795	0.3476	0.1196	54.28%	57.36%

Table 7: Time cost of BIKER, DeepAPI, and FIMAX.

Approach	Device	Training	Recommendation
BIKER	Intel i7-4790 CPU	N/A	5.4s/query
DeepAPI	Nvidia V100 GPU	14 hrs	0.57s/query
FIMAX	Intel i7-4790 CPU	3 mins	0.00576s/query

DeepAPI+FIMAX declines -7.99% compared to the performance of DeepAPI. One of the possible reasons for this is that the Scikit-learn library contains many statistical algorithms that are rarely used and queried on SO, making it difficult for FIMAX to learn the correct API usage patterns while returning many false positives, which further causes a performance decline.

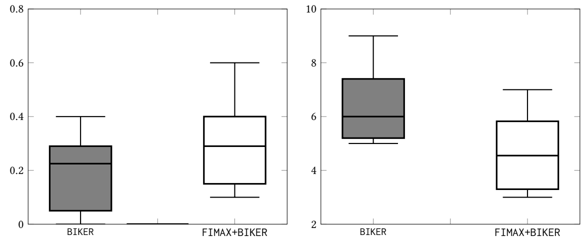
Table 7 shows the time cost of FIMAX and the baselines. In the training phase, BIKER requires no training, and DeepAPI requires 14 hours to train the model. In the recommendation phase, BIKER requires 5 seconds, while DeepAPI requires 0.5 seconds for each query. The time cost of FIMAX is only 3 minutes in training and 0.00576 seconds per query in the recommendation phase. Compared to the time cost of BIKER, the time cost of FIMAX is negligible in both the training and recommendation phases.

6 USER STUDY

In this section, we conduct a user study to further investigate whether FIMAX can help developers find correct APIs more efficiently and accurately.

6.1 Study Design

To conduct our user study, we randomly selected 20 questions from our test dataset. For each selected question, we created an API retrieval task using the question title as the task description. We invited four PhD students and ten MS students familiar with machine learning development to complete the 20 tasks. The years of their experience in developing machine learning software based on Python varied from two to six years, with an average of 4 years. We then divided the participants into two groups (G1 and G2), with experience evenly distributed in both groups. The 20 tasks were also randomly divided into two groups (T1 and T2). The experiment was conducted in two phases. In the first phase, participants in G1 and G2 were asked to complete the tasks in T1 using FIMAX+BIKER and BIKER, respectively. In the second phase, the two groups exchanged tools to complete the tasks in T2. Each participant had to record his/her screen during the experiment so that we could record how much time he/she spent on a question. Note that, DeepAPI is not evaluated since it performs relatively poor than BIKER.



(a) Correctness **(b) Completion Time (mins)**

Figure 2: Results of user study.

6.2 Results Analysis

Following existing studies [19, 45], we use two metrics to measure the performance of the participants on API retrieval task, i.e., **correctness** and **completion time**. Specifically, correctness evaluates whether a participant can find the correct APIs for a given question, and we measure the proportion of correct APIs submitted by a participant among all APIs in the ground truth answer of the question. Completion time evaluates how quickly a participant can answer a given question. For each question, we recorded the correctness and completion time of each participant, as well as the average value of the two groups of participants.

Figure 2 shows the performance of the groups with BIKER and FIMAX+BIKER over the 20 tasks. Using FIMAX+BIKER, participants completed the tasks more accurately and took less time than participants using the original BIKER. On average, the correctness and completion time (in minutes) of participants using FIMAX+BIKER and BIKER were 0.45 and 4.5, versus 0.33 and 6.3, respectively. We further used Wilcoxon signed-rank test for verifying the statistical significance of the differences. The p-values for both correctness and completion time are small than 0.05, which indicates that the differences of FIMAX+BIKER and BIKER in correctness and completion time are statistically significant.

7 DISCUSSION

7.1 More Reasons for Performance Decline

In section 4.3, we present two major reasons that contribute to the performance decline of existing API recommendation approaches.

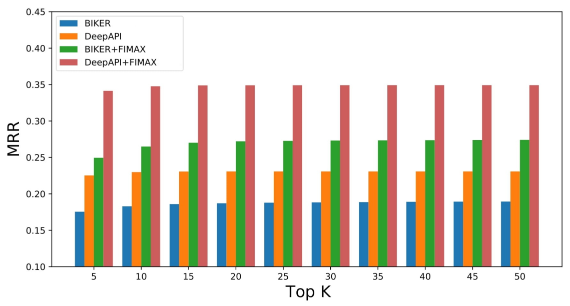


Figure 3: Performance of FIMAX under different top-k setting regarding MRR

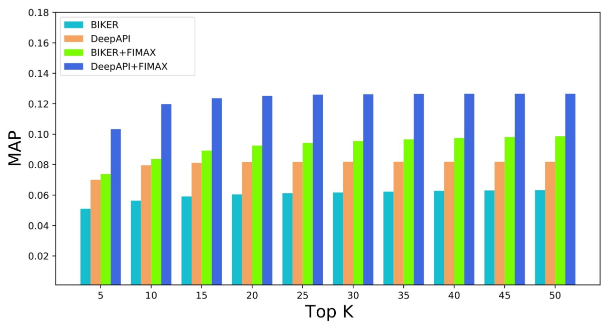


Figure 4: Performance of FIMAX under different top-k setting regarding MAR

In addition, there are several other reasons for the performance decline that we found in our investigation of the recommendation results, which are related to the nature of machine learning. We present these reasons as follows to motivate the future directions to improve the API recommendation on Python-based ML questions.
Library Bias: We note that some users specify the library name in their questions and others do not. When a user does not specify the library name, the API recommendation approach suffers from the popularity bias, i.e., the APIs of more popular libraries in the training dataset are more likely to be recommended. For example, the API used for a fully connected layer in TensorFlow is "*tf.contrib.layers.fully_connected*" while it is "*keras.layers.* Dense" in Keras. When a user asks questions about implementing a fully connected dense layer without mentioning the library name, the model tends to recommends "*keras.layers.Dense*" rather than "*tf.contrib.layers. fully_connected*". This is because there are many more questions about "*keras.layers.Dense*" than that of "*tf.contrib.layers.fully_connected*" on Stack Overflow, and rarely used APIs are less likely to be recommended. Such bias can be mitigated by specifying the library name in the question or balancing the training examples. This also motivates the practical need for fairness study in API recommendations [4, 15].
Multiple solutions: There are questions that have multiple solutions. For example, a question asking how to iterate through a Pandas column can be answered with solutions such as "*df.iterrows()*" or "*df.iteritems()*". Although both are correct recommendations, following the existing evaluation method [17, 19], only the recommendation that matches the answer in the collected dataset is considered correct. Such problems can be mitigated by expanding the answers to their synonyms, or collecting all the accepted answers. Moreover, this finding also indicates how inflexible the existing evaluation method for API recommendations is and motivates the need for new evaluation criteria.
Documentation issue: Another possible reason is that the format and organization of the documentation of Python-based ML libraries are different from those of Java JDK, which mainly affects the performance of BIKER. BIKER employs the documentation for similarity-based API re-ranking under the assumption that the documentation can provide description of the corresponding API. However, the API documentation of the Python ML libraries contains a detailed description of the input, output, caveats, and examples. This severely disturbs the recommendation model when calculating the similarity between the documentation and the question, resulting in performance degradation. This motivates the need for documentation understanding and key information extraction to automatically reorganize various sections of Python-based ML API documentation before applying BIKER.

7.2 What is the Performance of FIMAX Against Different Top-k Settings?

Both BIKER [19] and DeepAPI [17] can recommend top k APIs for a given question. Following their suggestion, in this work, we evaluate the top 10 recommended APIs in the API sequences recommended by the baseline methods (i.e., BIKER and DeepAPI) and also in the API sequences extended by both BIKER+FIMAX and DeepAPI+FIMAX. However, since the number of APIs required to solve a Python-based ML question is much larger than that of a Java JDK question (details are in Section 4.2), we further evaluate the performance of FIMAX under more recommended APIs, i.e., from 5 to 50 with 5 as the interval.

Figure 3 shows the MRR of BIKER, DeepAPI, BIKER+FIMAX and DeepAPI+FIMAX under different top-k configurations. We can see that the MRR values of BIKER and DeepAPI are below 0.20 and 0.25 from the top 5 to the top 50 recommendation results respectively. Also, the performance of BIKER and DeepAPI remains almost the same from the top 5 to the top 50 recommendations, which means that BIKER and DeepAPI cannot hit more correct APIs even if we include more recommendations.

After FIMAX is applied to both BIKER and DeepAPI, the performance of both approaches increases dramatically and remains stable across different top K settings. From Figure 3, we can see that the performance of BIKER+FIMAX is higher than BIKER at each top-K setting and increases slightly until the top-15 recommendation, meaning that BIKER+FIMAX is able to recommend the correct first match within 15 recommendation results. Figure 4 shows MAP values of BIKER, DeepAPI, BIKER+FIMAX, and DeepAPI+FIMAX under top k configurations from 5 to 50. Similar to the MRR result, the MAP values of BIKER and DeepAPI remain almost the same from the top 5 to the top 50. While the MAP values of BIKER+FIMAX and DeepAPI+FIMAX increase from top 5 to top 15, which shows that the FIMAX extension approach is able to increase

Table 8: Performance of FIMAX on different Java SO questions regarding the length of API sequence in the answers.

Java Questions	Improvement		
	MRR	**MAP**	**count**
Single Answer (=1)	3.2%	3.2%	100
Medium Answer ([2, 5])	0.8%	5.2%	100
Long Answer (>5)	0.4%	6.9%	100
Overall	1.0%	5.0%	300

the completeness of the original recommendation result of BIKER and DeepAPI in top k settings from 5 to 50. Also, BIKER+FIMAX and DeepAPI+FIMAX are able to recommend correct APIs not only for the first 10 but also for the first 15 recommendation results.

7.3 Generalizability of FIMAX

In this paper, FIMAX has been shown to be effective in improving the performance of API recommendation on Python-based ML programming tasks whose answer contains a long API call sequence, while its performance on traditional Java SO questions that require few APIs is unknown (details are in section 4.3.1). In this section, we further examine the generalizability of FIMAX by applying it to Java JDK questions. Specifically, for this experiment, we reuse the JAVA JDK dataset of BIKER [19]. FIMAX mines the API usage rules on the training dataset and tunes its parameters (i.e., support and confidence) by following the same process used in section 5.2. Since most Java questions require few APIs, we further divide the test dataset into three categories based on the number of APIs required, i.e., Single (requires one API), Medium (requires two to five APIs), and Long (requires more than five APIs). For the Single category, we randomly select 100 questions from the original test dataset of BIKER. Meanwhile, we find that most questions in BIKER's test dataset are single-answer questions, thus for Medium and Long categories, we randomly select 100 samples from the BIKER's training data and exclude them when training BIKER.

Table 8 shows the improvement of FIMAX+BIKER regarding MRR and MAP for different types of JAVA questions compared to the performance of BIKER. From the result, we can see that MAP increases by about 5% for the three categories and MRR improves by 3.2% for Single answer. The increase in MRR results from cases where none of the original API recommendations are correct, but the extended API hits the correct answer. The experiment result on Java JDK questions is consistent with that of Python-based ML questions, i.e., the FIMAX has a positive effect on the completeness of API recommendations. Our Wilcoxon signed-rank test result ($p < 0.05$) further confirms that FIMAX+BIKER could achieve significantly better performance than BIKER.

7.4 Threats to Validity

Internal Threat: Since BIKER and DeepAPI are designed for Java questions. We need to make necessary adaptation on them for Python-based ML questions. We reused the published source code of BIKER and DeepAPI for the baseline method with our dataset. We have carefully reviewed the implementation of FIMAX and the process of applying FIMAX to BIKER and DeepAPI to ensure that FIMAX works as intended. Therefore, the threat to internal validity is low.

Construct Threat: We follow the existing work [19] and adopt the same metrics (i.e., MRR and MAP) for performance evaluation. MRR and MAP are computed by reusing the algorithm in the source code of BIKER. However, the performance of FIMAX could be different if other metrics are used. In the future, we plan to examine FIMAX with other metrics, e.g., Precision and Recall, which are widely used in information retrieval [37].

External Threat: We collect the Python-based ML questions dataset from the official data explorer provided by Stack Exchange [3]. Although we follow the same criteria as BIKER to filter the noise data, our dataset could still contain noise. To reduce this threat, we manually checked our test dataset as described in section 4.1, the agreement among the authors measured with the Cohen's Kappa coefficient is 0.8731, indicating high agreement.

8 RELATED WORK

API Recommendation: There are many other approaches for API recommendation other than BIKER and DeepAPI [10, 25, 34, 35, 45]. Rahman et al. [35] proposed Rack for class-level Java API recommendation using a customized co-occurrence-based data-mining algorithm on top of the Lucien engine. BIKER reported better performance compared to RACK at the class level. Raghothaman et al. [34] proposed SWIM for synthesizing code snippets using API usage pattern mining techniques on GitHub code repositories. It trains a query-to-API model with the API call sequences from GitHub, and then synthesizes code snippets for a given query using the model. The difference between FIMAX and RACK and SWIM is that FIMAX applies the Apriori algorithm for mining API usage patterns, while Rack uses a customized technique and SWIM uses a mixture of several algorithms for code synthesis. McMillan et al. [25] proposed *Portfolio* for recommending related functions in C++ using data mining techniques in code archives. It applies the PageRank algorithm as well as the function call graph for association model building, and a customized similarity metric for relevant function ranking. Chan et al. [10] proposed a graph search approach that improves the performance of *Portfolio*.

They first build an API call graph using the text phrase in API-related questions, and then apply a customized shortest path indexing scheme for result ranking. Their approach significantly improves the performance of *Portfolio*.

Most existing studies on API recommendations have focused on Java programming questions. In this work, we investigate two state-of-the-art approaches, i.e., BIKE and DeepAPI, for Python-based ML programming tasks. He et al. [18] proposed PyART for real-time Python API recommendation. PyART is able to recommend not only APIs from third-party libraries, but also project specific APIs. We did not compare to PyART because we believe that such a comparison can be unfair, i.e., PyART and our approach use different information for API recommendation in different scenarios. Specifically, PyART focuses on real-time code completion and makes API recommendations based on the context of the programming tasks. Our approach is applied to a programming QA system by providing a search service that accepts a programming question and returns a list of APIs.

Mining API Usages: There are many studies on mining API usage [20, 27, 30, 33, 39, 51]. Treude et al. [39] proposed SISE, an API documentation augmentation technique that provides usage insights to developers. It applies a pattern-based approach with consideration of part-of-speech tags for feature extraction and a supervised machine learning approach for extracting insights from SO posts. In a comparative study with eight software developers, SISE was found to contribute the most useful information to API documentation. Moreno et al. [27] have proposed MUSE for mining code examples. Given a particular method, it returns a list of related code examples generated by a combination of code clone detection and static slicing, and ranks the result based on a set of usage-based heuristics rules. Zhong et al. [51] proposed MAPO for recommending related code snippet using frequent API usage patterns. MAPO builds a recommendation model using a frequent sub-sequence mining algorithm with source code extracted from Google code archives. For a given API method, it recommends the associated API calls based on the usage patterns captured by the model. Petrosyan et al. [33] proposed an approach to discover tutorial sections to explain a particular API type. They create a supervised text classification model for classifying tutorial fragments based on linguistic and structural features. Nguyen et al. [30] proposed API2VEC, a model that learns the semantic relationship between APIs using word embedding techniques. They build an embedding model using the CBOW word2vec model with the extracted API sequence from Java and C# code snippets. They demonstrate the usefulness of API2VEC with 3 example applications, including an example of a newly discovered API mapping between Java and C# code. They go one step further and develop an automated API mapping discovery tool called API2API based on API2VEC for API migration tasks between Java and C#. Jiang et al. [20] have proposed an unsupervised approach called FRAPT for finding relevant tutorial fragments for a given API. The difference between FRAPT and our approach is that FRAPT uses PageRank and a topic model-based algorithm, while FIMAX uses the Apriori algorithm for association rule mining.

9 CONCLUSION

In this paper, we investigate the effectiveness of state-of-the-art API recommendation approaches, i.e., BIKER and DeepAPI, on Python-based ML programming tasks. Specifically, we conducted an empirical study of programming questions related to six widely used Python-based ML libraries. We find two main reasons that contribute to the performance decline of existing approaches: (1) Python-based ML tasks often require significant long API sequences, and (2) there exist common API usage patterns in Python-based ML programming tasks that existing approaches cannot handle. Inspired by our findings, we proposed FIMAX, which enhances existing API recommendation approaches by using API usage information mined from SO questions. Our evaluation shows that FIMAX can significantly boost existing state-of-the-art API recommendation approaches. Our user study further demonstrates the practical value of FIMAX for API recommendations.

In the future, we plan to investigate the effectiveness of FIMAX on other API recommendation approaches and programming tasks in other domains and languages.

10 ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the National Natural Science Foundation of China under grant No.62072442, and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2021. *PyPI Download Stats.* https://pypistats.org/

[2] 2021. *Python Package Index - PyPI.* https://pypi.org/

[3] 2021. Query stackoverflow - Stack Exchange data explorer. Available at https://data.stackexchange.com/stackoverflow/query/new (2021/08/01).

[4] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 625–635.

[5] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Citeseer, 487–499.

[6] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International conference on machine learning.* PMLR, 2123–2132.

[7] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 291–300.

[8] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 513–522.

[9] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 628–632.

[10] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* 1–11.

[11] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering.* Springer, 385–400.

[12] Tapajit Dey, Andrey Karnauch, and Audris Mockus. 2021. Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 995–1007.

[13] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv preprint arXiv:1711.10604* (2017).

[14] Tomasz Drabas and Denny Lee. 2017. *Learning PySpark.* Packt Publishing Ltd.

[15] Sanghamitra Dutta, Dennis Wei, Hazar Yueksel, Pin-Yu Chen, Sijia Liu, and Kush Varshney. 2020. Is there a trade-off between fairness and accuracy? a perspective using mismatched hypothesis testing. In *International Conference on Machine Learning.* PMLR, 2803–2813.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

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

[18] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. PyART: Python API Recommendation in Real-Time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 1634–1645.

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

[20] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 38–48.

[21] Nikhil Ketkar. 2017. Introduction to keras. In *Deep learning with Python.* Springer, 97–111.

[22] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated*

Software Engineering (ASE). IEEE, 476–481.

[23] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[24] Wes McKinney. 2012. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.".

[25] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.

[26] Sandy Moens, Emin Aksehirli, and Bart Goethals. 2013. Frequent itemset mining for big data. In *2013 IEEE international conference on big data*. IEEE, 111–118.

[27] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 880–890.

[28] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 25–34.

[29] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. 2019. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52, 1 (2019), 77–124.

[30] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[31] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[33] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. 2015. Discovering information explaining API types using text classification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 869–879.

[34] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 357–367.

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

[36] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–355.

[37] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval.* Vol. 39. Cambridge University Press Cambridge.

[38] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic recommendation of API methods from feature requests. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 290–300.

[39] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 392–403.

[40] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2013. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *2013 International Conference on Social Computing*. IEEE, 188–195.

[41] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[42] Shaowei Wang, David Lo, and Lingxiao Jiang. 2013. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1019–1024.

[43] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. 2021. Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1548–1560.

[44] Xin Xia and David Lo. 2017. An effective change recommendation approach for supplementary bug fixes. *automated software engineering* 24, 2 (2017), 455–498.

[45] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1015–1026.

[46] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated generation of answer summary to developers' technical questions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 706–716.

[47] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. 2018. Domain-specific cross-language relevant question retrieval. *Empirical Software Engineering* 23, 2 (2018), 1084–1122.

[48] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE, 127–137.

[49] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 956–961.

[50] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.

[51] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.