BRAID: An API Recommender Supporting Implicit User Feedback

Yu Zhou
zhouyu@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics
Nanjing, China

Haonan Jin
jinhaonan@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics
Nanjing, China

Xinying Yang
xy_yang@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics
Nanjing, China

Taolue Chen
t.chen@bbk.ac.uk
Birkbeck, University of London
London, UK

Krishna Narasimhan
kri.nara@st.informatik.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Harald C. Gall
gall@ifi.uzh.ch
University of Zurich, Switzerland
Zurich, Switzerland

ABSTRACT

Efficient application programming interface (API) recommendation is one of the most desired features of modern integrated development environments. A multitude of API recommendation approaches have been proposed. However, most of the currently available API recommenders do not support the effective integration of user feedback into the recommendation loop. In this paper, we present *BRAID* (**B**oosting **R**ecommend**A**tion with **I**mplicit Fee**D**back), a tool which leverages user feedback, and employs learning-to-rank and active learning techniques to boost recommendation performance. The implementation is based on the VSCode plugin architecture, which provides an integrated user interface. Essentially, BRAID is a general framework which can accommodate existing query-based API recommendation approaches as components. Comparative experiments with strong baselines demonstrate the efficacy of the tool. A video demonstrating the usage of BRAID can be found at https://youtu.be/naD0guvl8sE.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**.

KEYWORDS

API recommendation, Learning to rank, Active learning, Natural language processing

ACM Reference Format:
Yu Zhou, Haonan Jin, Xinying Yang, Taolue Chen, Krishna Narasimhan, and Harald C. Gall. 2021. BRAID: An API Recommender Supporting Implicit User Feedback. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473111

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473111

1 INTRODUCTION

Application Programming Interfaces (APIs) are important building blocks to construct large software systems. A lot of API recommendation research work has been proposed to ease the burden on developers and provide APIs for user reference. With the help of APIs, developers can accomplish their programming tasks more efficiently. However, due to the huge number of APIs available for the same tasks, it is impractical for developers to get familiar with all of them and always select the correct ones for specific development tasks.

To aid with this difficult task of choosing APIs, a lot of API recommendation approaches have been proposed. Based on different input contexts, there are generally two types of recommendation scenarios, i.e., recommendation with queries and recommendation without queries. The first type requires developers to state their desired task using natural language queries which are fed into the recommendation system. For the second type, since there are no explicit queries, the surrounding code snippets will be leveraged as context, and the needed APIs will be inferred and recommended to end users. A majority of related work employs text similarity-based techniques. For example, some recommend APIs according to the similarity between search queries and supplementary information of APIs [4, 16]; some return API usages depending on how much they are related to context information in source code [2, 9]. Generally, these approaches use keywords to narrow down the search scale in massive target repositories and speed up recommendation efficiency. However, most of these approaches do not consider user interaction information in the recommendation process, such as the selection of certain APIs in the returned list. We believe this information is crucial to improve the API recommendation performance.

In this demo paper, we present BRAID[1] (**B**oosting **R**ecomm end**A**tion with **I**mplicit Fee**D**back), a tool that can recommend

[1]BRAID is open sourced at https://github.com/yyyxy/vscode-plugin-for-braid/

APIs to users by integrating their feedback information. The feedback denotes users' selection of returned APIs. The methodology underpinning BRAID is based on our previous work [18].

Different from traditional approaches, in BRAID, API recommendation is regarded as a ranking problem by optimizing an existing recommendation result. To this end, BRAID employs learning-to-rank (LTR) [5] and active learning techniques [13]. The key of LTR in information retrieval is to train a ranking model by which a given query can optimally order the relevant documents based on feedback. By viewing APIs as documents, we can apply LTR techniques to API recommendation to boost its performance. In particular, we leverage *related information features* and *feedback features* to train the model. The former consists of API path features and API description features, representing the relevance of the recommended APIs in relation to the user query and the associated document descriptions respectively; the latter represents the relevance to the APIs in the feedback repository.

Furthermore, to accelerate the feedback learning process, we incorporate active learning which is to alleviate the "cold start" problem. The active learning module can help achieve better performance with a relatively small amount of tenuous feedback information at the beginning. To this end, we leverage crowdsourced knowledge from Q&A websites, such as Stack Overflow[2], to extract questions and their answers. Based on this information, we can construct query-API pairs, where the query corresponds to the question, and the APIs correspond to the ones in the accepted answers. These pairs function as the oracle to provide the correct labels, and are then put to the training set. By iterating this process we can obtain a well-trained active learning model with the expanded labeled set. Due to space limitation, interested readers can refer to [18] for technical details.

As aforementioned, BRAID requires an initial recommendation list. To be flexible, BRAID is designed as a framework, allowing for third-party API recommenders to be plugged as a component. Developers can benefit from such a design decision, which implies BRAID is flexible and can be customized by different developers.

2 THE BRAID TOOL

BRAID is implemented as a VS Code[3] plug-in. VS Code was the most popular development environment as per a stack overflow survey [4]. Figure 1 illustrates the overview of the proposed approach.

The basic workflow of our approach is as follows.

(1) When a user makes a query Q to the system (in the form of, for instance, a short sentence in a natural language), a base API recommendation method is employed to provide an initial API list L_Q.
(2) The system looks up the feedback repository FR, checking whether or not there is a query similar to the user query Q. If this is the case, the system returns a set SP of query-API pairs where the similarity score of each query with Q is above a certain threshold. Otherwise, there is no available query in FR similar to Q (which is especially the case at the initial stage of the interaction), and SP is simply an empty set. The recommended APIs in L_Q and SP are to be fed to the feature extraction engine.
(3) The feature extraction engine, upon receiving L_Q and SP, computes a composite feature vector FV. FV includes two components, i.e., FF and RIF. The former corresponds to the feedback features, while the latter corresponds to the related information features, which are extracted from the related API documentation and Q&A posts. (In case that SP is empty, FV consists solely of related information features.)
(4) The ranking engine takes FV as input, and applies the trained learning-to-rank model and active learning model to obtain the prediction values. The system then calculates the API scores based on the prediction values of these two models. Afterwards L_Q is re-ranked in descending order according to the API scores, and new recommendations are presented to the users.

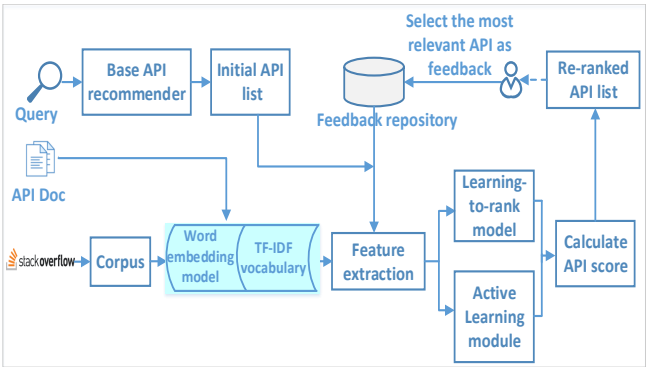

Figure 1: The overview of BRAID

Figure 2 and Figure 3 present the main user interfaces of BRAID. In Figure 2, a user can initiate the recommendation by simply a mouse right-click action in the edit area. Then a comprehensive menu list will be displayed (A). The user can select the "Input your query" item (B) on the menu to activate the query input dialog (C). For example, the query could be "How to sort". Given the input query, BRAID conducts the API recommendation, the workflow of which has been described as above. The result list (D) is displayed in the main part of Figure 3. The list is composed of recommended APIs and their associated descriptions. "Next" and "Previous" buttons (E) are in the right part of the panel to help navigate the list, since many APIs could be returned, and the number might exceed the display limit of a single page.

In Figure 3, the user can select a specific API by a mouse left-clicking the item on the returned list. A back-end daemon will record the selection decision and the related query, and update the feedback repository accordingly. The repository will provide accumulated information for the LTR model and active learning module, aiming to enhance recommendation experience for end users. As one might expect, with the accumulation of feedback, the performance of BRAID will become increasingly better (cf. Section 3).

[2] https://stackoverflow.com/
[3] https://code.visualstudio.com/
[4] https://insights.stackoverflow.com/survey/2019#development-environments-and-tools

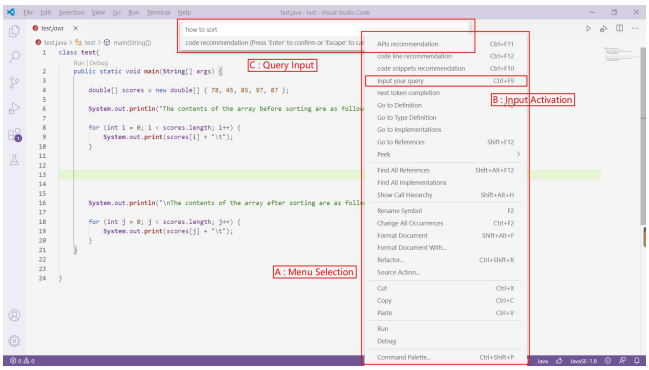

Figure 2: Query input UI of BRAID

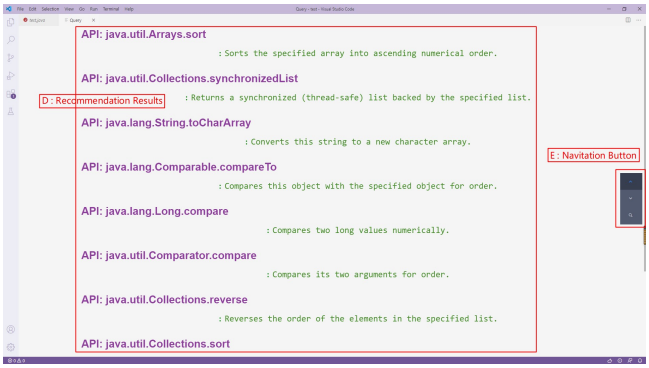

Figure 3: Recommendation result UI of BRAID

3 EVALUATION

In this section, we present the comparative experiment of BRAID against three state-of-the-art baselines, i.e., BIKER [4], RACK [12], and NLPAPI [10]. Due to space limitation, we mainly report our empirical results of the following research question, i.e., how does the accumulation of the feedback repository improve the performance of recommendation? This is the most important research question, since in real scenarios, the feedback repository is to be updated and accumulated from developers during programming over time. We are very interested in the contribution brought by the feedback information in BRAID.

To be fair, we reuse the datasets to construct the query-answer pairs, and the implementation to collect results, from the replication packages of the baselines. We follow the standard 10-fold cross validation and repeat the experiments 5 times. The average values are calculated as the final results. The experiments are conducted on a PC running Windows 10 OS with an AMD Ryzen 5 1600 CPU (6 cores) of 3.2GHz and 8GB DDR4 RAM. We randomly select the query-answer pairs from the training set to form the feedback repository. The size of the feedback repository varies from 0% to 100% of the training set, with an increment of 10%. Note that the baseline is represented by the case of size equal to 0%, where the feedback repository is empty or disabled.

Table 1 presents the experimental results with the three baseline recommenders as components in BRAID. The 'original' column lists the results of these baseline recommenders without the augmentation of feedback information. We can observe that the performance improves with the accumulation of the feedback repository. This is consistent across all the three baselines. All the metrics have been enhanced considerably. The MAP and MRR are 6% up for BIKER, over 13% up for RACK and NLP2API when 100% of the feedback repository is used.

Particularly, the most important indicator Hit@1 enjoys the largest boosting. Hit@1 is increased by 9.44% for BIKER , by 18% for RACK, and by 18.39% for NLP2API. Moreover, we use the Mann-Whitney U test and Vargha and Delaney's $\hat{A}_{12}$ statistic to examine these experimental results. Most $p-$values are in the range of 0.003 to 0.005, with effect size 1, indicating that the improvements are statistically significant at the confidence level of 99%. However, for BIKER there were 2 cases (metrics Hit@3 and hit@5 for 10% size of feedback repository) out of 50 where the $p-$values were higher than 0.005 (i.e., the null hypothesis should be rejected). For NLP2API, there was also one case (i.e., metrics Hit@5 for 10% size of feedback repository) where the p-value is higher than 0.005. We suspect that, when the feedback information is insufficient, our approach may not bring significant improvement on certain occasions. However, with the growth of feedback, our approach does show significant improvement over the baselines.

To further demonstrate how the user is involved and the effectiveness of our approach, we conduct an additional experiment where we consider a pseudo-user. We randomly select 50 queries, and the pseudo-user is programming during which the 50 queries are to be made. During each query, BRAID recommends APIs based on the feedback repository, and the pseudo-user selects API(s). The query and selected API(s) are used to expand the feedback repository. We train the models as soon as the feedback repository is not empty. The model is not re-trained during the 50 queries. Table 2 shows the results for pseudo-user experiment. The conclusion is consistent with other experiments that the results of Hit@1 metric improve the most, i.e., Hit@1 increase for BIKER is around 6%, for NLP2API is around 5%, and for RACK is over 9%.

4 RELATED WORK

A majority of the relevant work rely on code searching techniques to find the most similar ones and recommend to develoers. Examples include Strathcona [3], Portfolio [7], SENSORY [1], and Aroma [6]. Strathcona recommends code examples for developers by comparing structural similarity in the code repository; Portfolio mainly combines NLP, and spreading activation network algorithms to find the most relevant code for users; SENSORY considers the statement sequence information and uses the Burrows-Wheeler Transform algorithm to search in the code repository; Aroma takes a partial code snippet as query input, and returns a set of code snippets as recommendations. The above approaches mainly rely on code information to perform recommendation.

Meanwhile, some approaches employ additional information from other software artifacts or crowdsourced knowledge. Examples include BIKER [4], RACK [12], and NLP2API [10], all of which serve as our baselines in this paper. These approaches leverage Q&A posts from Stack Overflow website to find the most relevant APIs. NLP2API also incorporates (pseudo-) feedback information

Table 1: The effect of BRAID with accumulation of the feedback information

Baseline	Metric	Original	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
BIKER	Hit@1	**0.4231**	0.4418	0.4704	0.4931	0.4986	0.5020	0.5073	0.5112	0.5146	0.5170	**0.5175**
	Hit@3	**0.6607**	0.6815	0.7018	0.7140	0.7178	0.7178	0.7193	0.7203	0.7203	0.7208	**0.7213**
	Hit@5	**0.7747**	0.7825	0.7945	0.8024	0.8062	0.8067	0.8072	0.8077	0.8091	0.8096	**0.8110**
	MAP	**0.5534**	0.5689	0.5919	0.6072	0.6106	0.6128	0.6155	0.6176	0.6205	0.6214	**0.6223**
	MRR	**0.5685**	0.5816	0.6035	0.6189	0.6226	0.6252	0.6282	0.6308	0.6334	0.6346	**0.6356**
RACK	Hit@1	**0.3267**	0.4160	0.4587	0.4827	0.4840	0.4893	0.4907	0.4947	0.5000	0.5040	**0.5067**
	Hit@3	**0.5133**	0.5680	0.5933	0.6013	0.6120	0.6133	0.6147	0.6173	0.6200	0.6360	**0.6400**
	Hit@5	**0.6267**	0.6453	0.6640	0.6667	0.6720	0.6733	0.6733	0.6773	0.6813	0.6813	**0.6867**
	MAP	**0.4203**	0.4789	0.5211	0.5345	0.5418	0.5434	0.5434	0.5490	0.5538	0.5588	**0.5620**
	MRR	**0.4506**	0.5120	0.5455	0.5622	0.5654	0.5675	0.5692	0.5722	0.5765	0.5819	**0.5852**
NLP2API	Hit@1	**0.3516**	0.3871	0.4452	0.4761	0.4916	0.5181	0.5213	0.5226	0.5258	0.5310	**0.5355**
	Hit@3	**0.5323**	0.5561	0.5877	0.6039	0.6187	0.6284	0.6316	0.6342	0.6348	0.6348	**0.6355**
	Hit@5	**0.6000**	0.6187	0.6413	0.6426	0.6523	0.6555	0.6619	0.6626	0.6632	0.6645	**0.6645**
	MAP	**0.4111**	0.4451	0.4851	0.5123	0.5249	0.5408	0.5450	0.5480	0.5482	0.5524	**0.5549**
	MRR	**0.4604**	0.4885	0.5290	0.5502	0.5627	0.5807	0.5841	0.5867	0.5881	0.5912	**0.5937**

Table 2: Evaluation results for our framework pseudo-users experiments comparing with baselines

Baseline	Technique	Hit@1	Hit@3	Hit@5	MAP	MRR
BIKER	Original	0.4231	0.6607	0.7747	0.5534	0.5685
	Avg. BRAID	**0.4800**	**0.7000**	**0.8000**	**0.5924**	**0.5967**
	Abs. Imp.	5.69%	3.93%	2.53%	3.89%	2.82%
	Rel. Imp.	13.44%	5.94%	3.26%	7.04%	4.96%
RACK	Original	0.3267	0.5133	0.6267	0.4203	0.4506
	Avg. BRAID	**0.4200**	**0.6000**	**0.6600**	**0.5155**	**0.5410**
	Abs. Imp.	9.33%	8.67%	3.33%	9.53%	9.04%
	Rel. Imp.	28.57%	16.88%	5.32%	22.66%	20.06%
NLP2API	Original	0.3516	0.5323	0.6000	0.4111	0.4604
	Avg. BRAID	**0.4000**	**0.5600**	**0.6400**	**0.4643**	**0.5072**
	Abs. Imp.	4.84%	2.77%	4.00%	5.32%	4.68%
	Rel. Imp.	13.76%	5.21%	6.67%	12.93%	10.16%

as our work, but its purpose is to reformulate the query. Similarly, QUICKAR [11] also aims to automatically provide reformulation of a given query.

Some approaches augmented with other information for recommendation are APIREC [8], FOCUS [9], and LibraryGURU [17]. APIREC leverages fine-grained change commit history from Github to extract frequent change patterns to supplement the recommendation process. FOCUS tackles the usage pattern recommendation problem from the perspective of collaborative filtering, and similar projects information is consulted during the recommendation process. LibraryGURU combines code parsing and text processing on Android tutorials and SDK documents to recommend functional APIs in Android. However, none of these approaches consider effective integration of feedback into the recommendation.

There are a few work trying to leverage feedback into software engineering tasks. Sivaraman et al. [14] employ user feedback to express and refine search queries. Wang et al. [15] incorporate the feedback into the code search process and propose an active code search approach. However, both work address the code search problem instead of API recommendation.

5 CONCLUSION

In this paper, we have demonstrated BRAID, the tool we designed and implemented to automatically recommend APIs to developers. The primary feature of BRAID is its ability to incorporate user feedback information into the recommendation loop. We evaluated BRAID against three strong baselines. The results indicate that BRAID can effectively improve the performance of recommendation by integrating user feedback. Currently, we mainly support Java API recommendation, but extending to other programming languages is included in our future plan.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 61972197), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Qing Lan Project. T. Chen is partially supported by Birkbeck BEI School Project (ARTEFACT), NSFC grant (No. 61872340), and Guangdong Science and Technology Department grant (No. 2018B010107004).

REFERENCES

[1] Lei Ai, Zhiqiu Huang, Weiwei Li, Yu Zhou, and Yaoshen Yu. 2019. SENSORY: Leveraging Code Statement Sequence Information for Code Snippets Recommendation. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 27–36. https://doi.org/10.1109/COMPSAC.2019.00014

[2] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 254–265. https://doi.org/10.1145/2950290.2950319

[3] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 117–125. https://doi.org/10.1109/ICSE.2005.1553554

[4] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 293–304. https://doi.org/10.1145/3238147.3238191

[5] Tie-Yan Liu. 2011. *Learning to rank for information retrieval.* Springer Science & Business Media. https://doi.org/10.1007/978-3-642-14267-3

[6] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the*

ACM on Programming Languages 3, OOPSLA (2019), 152. https://doi.org/10.1145/3360578

[7] 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*. ACM, 111–120. https://doi.org/10.1145/1985793.1985809

[8] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. 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*. ACM, 511–522. https://doi.org/10.1145/2950290.2950333

[9] Phuong Nguyen, Juri Di Rocco, Davide Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. Focus: A recommender system for mining api function calls and usage patterns. In *41st ACM/IEEE International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00109

[10] Mohammad Masudur Rahman and Chanchal Roy. 2018. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 473–484. https://doi.org/10.1109/ICSME.2018.00057

[11] Mohammad Masudur Rahman and Chanchal K Roy. 2016. QUICKAR: automatic query reformulation for concept location using crowdsourced knowledge. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 220–225. https://doi.org/10.1145/2970276.2970362

[12] Mohammad M 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. https://doi.org/10.1109/SANER.2016.80

[13] Burr Settles. 2009. *Active learning literature survey*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[14] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 292–303. https://doi.org/10.1109/ICSE.2019.00044

[15] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 677–682. https://doi.org/10.1145/2642937.2642947

[16] Haibo Yu, Wenhao Song, and Tsunenori Mine. 2016. APIBook: an effective approach for finding APIs. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*. ACM, 45–53. https://doi.org/10.1145/2993717.2993727

[17] Weizhao Yuan, Hoang H Nguyen, Lingxiao Jiang, Yuting Chen, Jianjun Zhao, and Haibo Yu. 2019. API recommendation for event-driven Android application development. *Information and Software Technology* 107 (2019), 30–47. https://doi.org/10.1016/j.infsof.2018.10.010

[18] Yu Zhou, Xinying Yang, Taolue Chen, Zhiqiu Huang, Xiaoxing Ma, and Harald C Gall. 2021. Boosting API recommendation with implicit feedback. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3053111