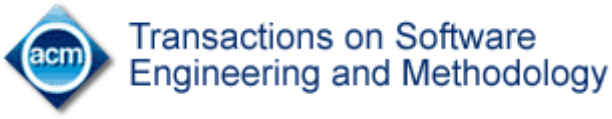


DDASR: Deep Diverse API Sequence Recommendation

Journal:	*Transactions on Software Engineering and Methodology*
Manuscript ID	TOSEM-2024-0012
Manuscript Type:	Paper
Date Submitted by the Author:	10-Jan-2024
Complete List of Authors:	Nan, Siyu; Wuhan University, School of Computer Science Wang, Jian; Wuhan University, School of Computer Science Zhang, Neng; Sun Yat-Sen University, School of Software Engineering Li, Duantengchuan; Wuhan University, School of Computer Science Li, Bing; Wuhan University, School of Computer Science
Computing Classification Systems:	API languages, Automatic programming

Dear Editor-in-Chief,

We would like to submit the manuscript entitled: “DDASR: Deep Diverse API Sequence Recommendation”. All authors have read this manuscript and would like to have it considered exclusively for publication in ***ACM Transactions on Software Engineering and Methodology.*** None of the material related to this manuscript has been published or is under consideration for publication elsewhere, including on the Internet.

The research background of this manuscript is described as follows. Recommending API sequences is an essential task in software development since it is time-consuming and labor-intensive for developers to inquire about APIs from massive API libraries. While previous studies primarily focus on accuracy, often recommending popular APIs, they tend to overlook less frequent, or `tail,' APIs. This oversight is often a result of limited historical data. Nevertheless, tail APIs are usually indispensable in large-scale software applications. Ignoring tail APIs would impairs the diversity of recommender systems. In this paper, we propose DDASR, a framework for recommending API sequences by considering both popular and tail APIs in response to a developer query. To accurately capture developer intent, we utilize recent Large Language Models for learning query representations. To gain a better understanding of tail APIs, DDASR clusters tail APIs with similar functionality and replaces them with cluster centers to produce a pseudo ground truth. Moreover, due to the inherent trade-off between accuracy and diversity, a loss function is defined based on learning-to-rank to achieve an equilibrium in accuracy and diversity. To evaluate DDASR, we conduct extensive experiments on two open-source datasets: a Java dataset and a Python dataset. Results demonstrate that DDASR significantly achieves the best diversity without sacrificing accuracy. Compared to four state-of-the-art approaches, DDASR improves accuracy metrics BLEU, MAP, and NDCG and diversity metric coverage by 108.28%, 88.59%, 207.37%, and 45.83%, respectively on the Java dataset, as well as 10.76%, 8.06%, 8.93%, and 8.03%, respectively on the Python dataset.

The main contributions of this manuscript are outlined as follows:

1) We propose DDASR, a novel framework for recommending API sequences,

striking a balance between accuracy and diversity.

2) We focus on the long-tail distribution in API sequence recommendation. We cluster tail APIs with functional similarity and construct a pseudo ground truth by replacing tail APIs in the original ground truth with cluster centers, effectively mitigating the sparsity of historical usage data for tail APIs.

3) We conduct extensive experiments to evaluate DDASR using open-source Java and Python datasets. A diverse Java dataset is also constructed for further evaluation. The experimental results show that our approach achieves significant diversity without reducing accuracy.

If you have any questions regarding this manuscript, please feel free to contact me.

Best regards,

Dr. Jian Wang

Associate Professor, School of Computer Science

Wuhan University, China

DDASR: Deep Diverse API Sequence Recommendation

SIYU NAN, School of Computer Science, Wuhan University, China
JIAN WANG*, School of Computer Science, Wuhan University, China
NENG ZHANG, School of Software Engineering, SUN Yat-sen University, China
DUANTENGCHUAN LI, School of Computer Science, Wuhan University, China
BING LI*, School of Computer Science, Wuhan University, China

Recommending API sequences is crucial in software development, saving developers time and effort. While previous studies primarily focus on accuracy, often recommending popular APIs, they tend to overlook less frequent, or 'tail,' APIs. This oversight, often a result of limited historical data, consequently diminishes the diversity of recommender systems. In this paper, we propose DDASR, a framework for recommending API sequences containing both popular and tail APIs. To accurately capture developer intent, we utilize recent Large Language Models for learning query representations. To gain a better understanding of tail APIs, DDASR clusters tail APIs with similar functionality and replaces them with cluster centers to produce a pseudo ground truth. Moreover, a loss function is defined based on learning-to-rank to achieve an equilibrium in accuracy and diversity due to the inherent trade-off between them. To evaluate DDASR, we conduct extensive experiments on Java and Python open-source datasets. Results demonstrate that DDASR significantly achieves the best diversity without sacrificing accuracy. Compared to four state-of-the-art approaches, DDASR improves accuracy metrics BLEU, MAP, and NDCG and diversity metric coverage by 108.28%, 88.59%, 207.37%, and 45.83%, respectively on the Java dataset, as well as 10.76%, 8.06%, 8.93%, and 8.03%, respectively on the Python dataset.

CCS Concepts: • **Software and its engineering → API languages**; **Automatic programming**.

Additional Key Words and Phrases: API Sequence Recommendation, Long-tail Distribution, Clustering, Diversity

ACM Reference Format:
Siyu Nan, Jian Wang, Neng Zhang, Duantengchuan Li, and Bing Li. 2024. DDASR: Deep Diverse API Sequence Recommendation. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2024), 29 pages.

1 INTRODUCTION

An application programming interface (API) is a software interface that encapsulates underlying functionalities. Reusing APIs can significantly enhance development efficiency and reduce associated costs. However, with the proliferation of APIs, developers often find it challenging to familiarize themselves with all APIs in the libraries. A survey by Ko et al. [30] indicates that selecting the appropriate API poses a significant learning barrier in user programming systems.

*The Corresponding Authors.

Authors' addresses: Siyu Nan, siyunan@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China, 430072; Jian Wang, jianwang@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China, 430072; Neng Zhang, zhangn279@mail.sysu.edu.cn, School of Software Engineering, SUN Yat-sen University, Zhuhai, Guangdong, China, 519082; Duantengchuan Li, dtclee1222@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China, 430072; Bing Li, bingli@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China, 430072.

Manuscript submitted to ACM

Query:
Simulate mouse wheel scrolling to control the scrolling of a webpage.

Code snippet:

```
public class MouseActionsExample {
    public static void main(String[] args) {
        Robot robot = new Robot();
        robot.mousePress(InputEvent.BUTTON1_MASK);
        robot.mouseRelease(InputEvent.BUTTON1_MASK);
        robot.mouseWheel(3);}}
```

API sequence:

① Robot.init
② Robot.mousePress
③ Robot.mouseRelease
④ Robot.mouseWheel

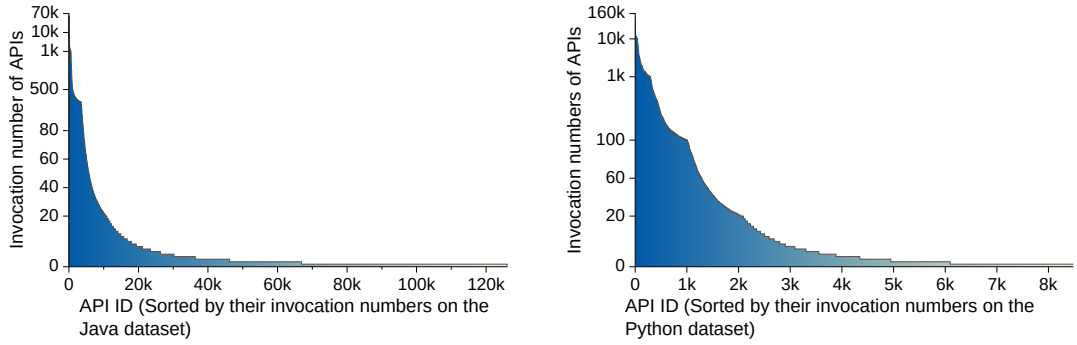
Fig. 1. An example query that needs to be addressed using an API sequence in Java.

Moreover, fulfilling user requirements often necessitates more than just a single API. Developers typically need to look up API sequences to solve tasks. For instance, to address a query like '*simulating mouse wheel scrolling to control the scrolling of a webpage*,' as shown in Fig. 1, a sequence of four APIs, including 'Robot.init,' 'Robot.mousePress,' 'Robot.mouseRelease,' and 'Robot.mouseWheel,' needs to be invoked.

It is a challenging problem to find appropriate API sequences from the vast array of APIs according to developer requirements [50]. Recently, several approaches have been proposed to recommend APIs for developers. These approaches fall into two major categories: information retrieval-based approaches [24, 48, 62] that search for the most relevant solutions from the historical question repository, and deep learning-based approaches [12, 19, 40] that adopt the sequence-to-sequence (Seq2Seq) model to recommend API sequences generatively. Gu et al. [19] adopt an RNN encoder-decoder and Elnaggar et al. [12] adopt a Transformer encoder-decoder to obtain the results of the API sequence. Martin and Guo [40] apply CodeBERT [14] for the task due to the improved performance of Large Language Models (LLMs). However, existing API recommendation approaches usually recommend popular APIs, neglecting less frequently used ones. For example, deep learning-based approaches [12, 19, 40] remove infrequently appearing words from the vocabulary or treat them as *<UNK>* tags, making it challenging to recommend infrequently occurring APIs to developers.

The long-tail effect [4, 45] occurs when APIs with low individual frequencies collectively constitute a substantial portion. As shown in Fig. 2, the long-tail distribution that is objectively present in API libraries can be observed in two open-source datasets: a Java dataset [19] and a Python dataset [40]. In the two datasets, APIs occurring twice or less are categorized as tail APIs, whereas the rest are non-tail APIs. This distinction aligns with the previous deep learning-based approaches [12, 19, 40]. There is a similar distribution pattern in Fig. 2(a) and Fig. 2(b). Frequently occurring non-tail APIs are clustered in a small portion at the head, while tail APIs, despite their low individual occurrence frequency, cover a large scale in quantity. From a cognitive perspective, exclusively recommending non-tail APIs may reduce the sense of novelty for developers [60], as non-tail APIs are already familiar to the majority of developers involved in software development. In addition, tail APIs can be the key to resolving specific queries for developers [30]. For instance, in Fig. 1, '*Robot.mouseWheel*' is a tail API that can be invoked to simulate mouse wheel scrolling functionality and is a vital component to solving the query. Hence, learning the representation of tail APIs is essential for API recommendation. However, it is a challenge due to the sparse historical usage data of tail APIs.

Moreover, ignoring tail APIs can harm the diversity of a recommendation system. Our focus is on 'aggregate diversity,' which refers to how many different APIs are included in the result set of API sequences generated by the recommendation system [1, 2, 4, 29, 58]. Recommending tail APIs can boost the visibility of those rarely used APIs and increase the aggregate diversity of recommendation results, thus improving the health of the overall software ecosystem. In addition to expanding the variety of APIs, the balance between the accuracy and diversity of the recommender



(a) (b)

Fig. 2. Long-tail distribution of APIs in the datasets. (a) Distribution in the Java dataset, and (b) distribution in the Python dataset.

system should also be considered. As a result, both non-tail and tail APIs must be represented in the recommendation results. However, accuracy and diversity are contradictory metrics, and balancing them becomes another challenge.

To address these issues, we propose a **d**eep **d**iverse **A**PI **s**equence **r**ecommendation framework (DDASR), which can recommend both non-tail and tail APIs and increase diversity without sacrificing accuracy. To better capture developer requirements, we leverage the recent advancements in LLMs. Due to the lack of historical data for tail APIs, their features cannot be adequately acquired. To alleviate the sparsity, we cluster tail APIs with a similarity matrix and substitute them with cluster centers. This strategy helps us establish a pseudo ground truth that includes both non-tail and tail APIs. The similarity matrix is built by calculating the similarity through the description and name of tail APIs. In addition, given that accuracy and diversity are two contradicting indicators, a loss function based on the learning-to-rank (LTR) technique is defined to optimize the ranking of recommendations while maintaining a balance between accuracy and diversity.

To demonstrate the effectiveness of DDASR, four state-of-the-art API recommendation approaches, i.e., DeepAPI [19], BIKER [24], CodeBERT [40], and CodeTrans [12], are selected as baselines, with BLEU, MAP, and NDCG as accuracy evaluation metrics, and coverage as the diversity evaluation metric. Two open-source datasets, a Java [19] dataset and a Python [40] dataset, as well as a diverse Java dataset derived from the Java dataset, are employed for evaluation. We evaluate the performance of DDASR utilizing three architectures: RNN encoder-decoder, Transformer encoder-decoder, and LLM encoder-decoder. The RNN and Transformer encoder-decoder architectures aim to reproduce the approaches proposed by Gu et al. [19] and Elnaggar et al. [12]. For the LLM encoder-decoder model, we use the recent five LLMs, such as CodeBERT [14] and GraphCodeBERT [21]. Overall performance is the best when using CodeBERT. In terms of accuracy metrics BLEU, MAP, and NDCG, DDSAR outperforms the baselines by an increase of 63.90%, 46.70%, and 128.60%, respectively on the original Java dataset, 108.28%, 88.59%, and 207.37%, respectively on the diverse Java dataset, and 10.76%, 8.06%, and 8.93%, respectively on the Python dataset. Regarding the diversity metric coverage, DDSAR surpasses the baselines by an increase of 45.83% on the diverse Java dataset and 8.03% on the Python dataset. The

results indicate that DDASR can significantly increase diversity in recommended API sequences without compromising accuracy. The replication package of DDASR is released at GitHub[1].

The main contributions of our work are as follows:

- We propose DDASR, a novel framework for recommending API sequences, striking a balance between accuracy and diversity.
- We focus on the long-tail distribution in API sequence recommendation. We cluster tail APIs with functional similarity and construct a pseudo ground truth by replacing tail APIs in the original ground truth with cluster centers, effectively mitigating the sparsity of historical usage data for tail APIs.
- We conduct extensive experiments to evaluate DDASR using open-source Java and Python datasets. A diverse Java dataset is also constructed for further evaluation. The experimental results show that our approach achieves significant diversity without reducing accuracy.

The rest of the paper is organized as follows. We present related work in Section 2. We describe the technical details of DDASR in Section 3. We describe our experimental setup and the evaluation results in Section 4. We discuss threats to validity in Section 5. Finally, we conclude the paper and put forward future work in Section 6.

2 RELATED WORK

Recommender systems have been extensively studied and applied in software engineering to aid developers in resolving issues of development requirements [18, 50]. This section mainly discusses the literature on recommender systems closely related to this paper, including API recommendation, sequence recommendation, and diverse recommendation.

2.1 API Recommendation

API recommendations generally start from developers' requirement queries. Recent research in API recommendation primarily relies on two frameworks: information retrieval and deep learning.

Information Retrieval-based Approaches. API recommendation approaches based on information retrieval typically perform similarity calculations from the historical library to match existing solutions. Zhong et al. [67] proposed MAPO to mine API usage patterns with sequential rules between APIs. McMillan et al. [41] proposed Portfolio, a tool for finding relevant functions based on PageRank from an extensive archive of C/C++ source code. Chan et al. [8] improved Portfolio using the API graph search algorithm. Raghothaman et al. [47] presented SWIM, an approach designed to find APIs related to the query from user click data and then locate API sequences by extracting and matching structured API sequences from GitHub. RACK [48] recommends APIs by exploiting keyword-API associations from Stack Overflow. Zhang et al. [62] proposed RASH to recommend APIs based on API specifications and the resolution of historical questions. Huang et al. [24] proposed BIKER, which combines historical questions and answers in Stack Overflow with API descriptions. However, information retrieval-based approaches struggle to capture the relevance of tail APIs.

Deep Learning-based Approaches. Deep learning-based generative API recommendation may produce creative results [34]. Niu et al. [43] extracted multiple features and employed learning-to-rank to recommend APIs and code examples. DeepAPI [19] builds an open-source dataset and customizes the RNN encoder-decoder neural language model for API sequence recommendation. Martin and Guo [40] developed a Transformer-based neural architecture with CodeBERT [14], an encoder-only pre-trained model, for API sequence learning. CodeTrans [12] uses both supervised

[1] http://github.com/nnroy/DDASR

and self-supervised tasks to build a language model, which has been applied to API sequence recommendation with the Transformer encoder-decoder architecture. Unfortunately, models such as DeepAPI and CodeTrans often split a complete API into multiple word or symbol fragments, potentially resulting in incorrect API recommendations.

2.2 Sequence Recommendation

A solution to a query involving development requirements often consists of a series of APIs. Classic sequence recommender systems adopt a Markov chain [49] to mine frequent patterns in sequence data. With the development of neural networks, RNN and its variants, such as Gated Recurrent Unit (GRU) [22], Long Short Term Memory (LSTM) [57], and hierarchical RNN [46], are recognized as beneficial for sequence recommendation. A few works [52, 61] applied Convolutional Neural Networks (CNN) to sequence recommendation. By adapting attention and self-attention mechanisms, researchers have recently developed many approaches, including NARM [36], SASRec [28], and ATTRec [64]. Recently, researchers have begun exploring the use of LLMs for sequence recommendation tasks [15]. LLMs can be adapted to user data collection, feature engineering, and feature encoder [38]. When applying LLMs, it is important to consider whether to fine-tune them during the training phase. For instance, models like TransRec [16], UNBERT [63], and LMRecSys [65], benefit from fine-tuning during the training phase. In contrast, models such as ChatGPT [11, 23, 39, 51] are designed to operate without the necessity for task-specific fine-tuning during the training phase.

2.3 Diverse Recommendation

Typical approaches for enhancing recommendation diversity can be classified into three categories: pre-processing, in-processing, and post-processing.

Pre-processing Approaches intervene in the recommendation system before the model training. DCF [10] regards the diverse recommendation as an end-to-end supervised learning task and constructs ground truth labels to explicitly idealize the optimization target. ART [33] leverages a strategy of pre-defining user types to enhance the diversity of recommendations. DGCN [66] includes two pre-defined sampling strategies for diversified recommendations with Graph Convolutional Networks (GCN). In addition, the studies [44, 60] utilized long-tail items directly for recommendations. Kim et al. [29] clustered long-tail items to predict a ranked list containing general and diverse items in the next item recommendation.

In-processing Approaches are applied during the model training process. Wasilewski and Hurley [55] explored the use of regularisation to enhance the diversity of recommendations. Li et al. [37] utilized factorized category features based on matrix factorization. Zhou et al. [68] adapted the Simpson's Diversity Index and considered the evenness of the number of the items' classes.

Post-processing Approaches re-rank recommended items based on certain diversity metrics. MMR [7] uses a model that treats relevance and diversity as independent metrics. Ziegler et al. [69] presented topic diversification to balance and diversify recommendation results. Adomavicius and Kwon [1] eliminated variations in popularity by assigning different weights to items. Jiang et al. [26] explicitly modeled subtopics to retrieve diverse results. DDP [9, 56] employs a unified model to assess differences among items in a set-wide way.

3 APPROACH

Fig. 3 shows the overall framework of DDASR, which consists of four main components:

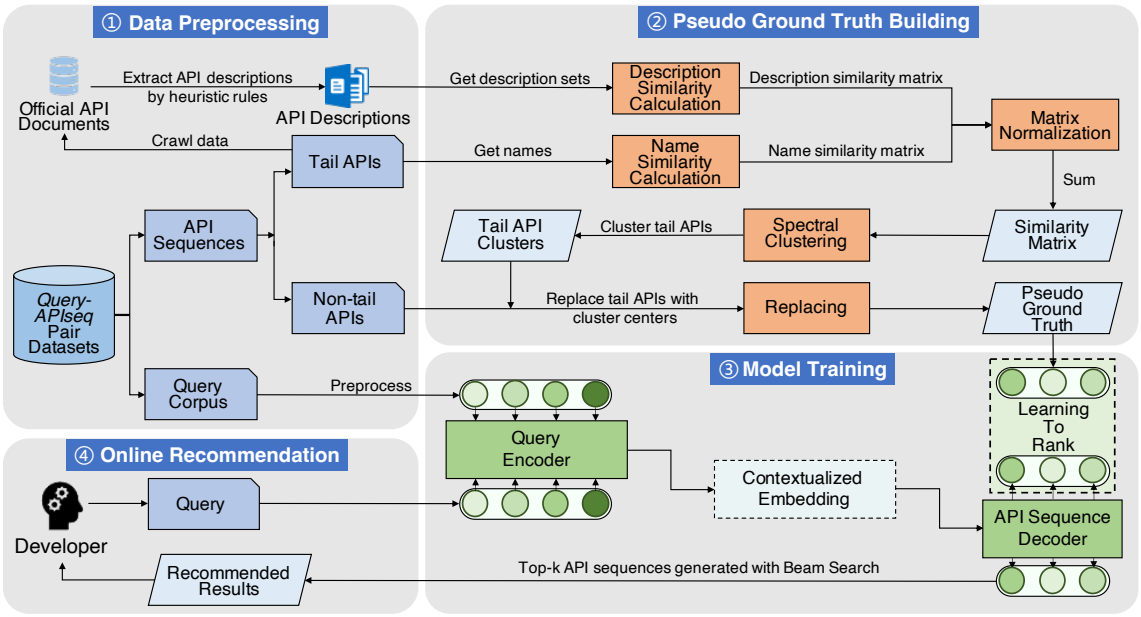


Fig. 3. Overall framework of DDASR.

(1) **Data Preprocessing** (cf. Section 3.1). We obtain *query-APIseq* pairs, which contain queries and corresponding API sequences. The non-tail and tail APIs are determined by figuring out how frequently they occur. Moreover, we crawl descriptions of tail APIs from the official API documents.

(2) **Pseudo Ground Truth Building** (cf. Section 3.2). We calculate the functional similarity among tail APIs with their descriptions and names. A pseudo ground truth, which contains both non-tail and tail APIs, is established by clustering tail APIs and substituting them with cluster centers.

(3) **Model Training** (cf. Section 3.3). Our model applies two techniques: Seq2Seq (cf. Section 3.3.1) and LTR (cf. Section 3.3.2). The Seq2Seq model is used to translate a given natural language query to a ranked list of possible API sequences. LTR calculates a loss function and ranks these API sequences. We integrate LTR within the Seq2Seq architecture to balance accuracy and diversity in the recommended API sequences. Lastly, Beam Search [31] is utilized to suggest relevant API sequences for developer queries.

(4) **Online Recommendation**. When developers input a query, DDASR can recommend an appropriate API sequence solution.

3.1 Data Preprocessing

For queries, we use the NLTK package [6] to deactivate stop words and extract the backbone of query terms. Furthermore, in the preprocessing of previous generation-based studies [12, 19, 40], an API is typically divided into multiple fragments. Each fragment is a symbol or word representing the class or method name. For example, '*ArrayList<String>.append*,' an API in Java programming language, will be split into six fragments: '*ArrayList*,' '<,' '*String*,' '>,' '.,' and '*append*.' Such division in preprocessing lengthens the API sequence, complicating the modeling of the API calling relationship and dramatically increasing computational costs. Furthermore, when recommending APIs to developers, this strategy also

results in mismatched fragments. Inspired by the word embedding technique [5], we recombine these fragments into complete APIs, allowing the model to better learn the calling relationship between APIs and reducing the computation of decoding APIs. After preprocessing the queries and API fragments, the corpus containing *query-APIseq* pairs for Java and Python are obtained. Subsequently, we conduct a frequency analysis of API occurrences and distinguish between non-tail and tail APIs.

For the tail APIs, we mine their descriptions to facilitate functional similarity calculations. To achieve this, we download both official API documentation and third-party documents. Then we parse the HTML file of each API class to extract their descriptions. Generally, APIs with inheritance relationships tend to have similar functions. Therefore, we also mine descriptions of APIs within these inheritance relationships as a supplementary measure.

3.2 Pseudo Ground Truth Building

Traditional API sequence recommendation approaches often overlook tail APIs, either by ignoring them or labeling them as *<UNK>* tags, which leads to these APIs being under-recommended. Noting that APIs with similar functions typically share analogous functionalities, we cluster tail APIs with similar functionality and then substitute them with cluster centers to build a pseudo ground truth. This pseudo ground truth includes both non-tail and tail APIs, ensuring comprehensive recommendations. In this context, we use TA and NA to represent tail APIs and non-tail APIs, respectively.

3.2.1 *Similarity Calculation.*

As a primary indicator, the name of an API typically reflects its function, while the official API documentation describes its functionalities. In this section, we calculate the functional similarity among tail APIs using both their documentation descriptions and names. A functional similarity matrix is ultimately constructed, containing the functional similarity between each pair of tail APIs.

Similarity calculation of API documentation descriptions. We observe that official documentation contains numerous inheritance associations among APIs. To augment the descriptions of child APIs, we append the official descriptions of their parent APIs. Designing APIs with a preference for shallow inheritance hierarchies is considered a good design principle and is widely accepted and recommended [17]. Therefore, we only consider API descriptions that involve inheritance relationships of parent nodes. Let $\mathbb{D}_i = \{D_{i,s}, D_{i,p}\}$ represent the description set of tail API TA_i, where $D_{i,s}$ is the description of TA_i itself, and $D_{i,p}$ denotes the description inherited from the parent node of TA_i.

We use BM25, an algorithm for evaluating the similarity between search statements and documents in the corpus, to calculate the description similarity between tail APIs. To assess the text-similarity between two descriptions, D_1 and D_2, we approach it in a bifurcated manner: $Sim(D_1 \rightarrow D_2)$ and $Sim(D_2 \rightarrow D_1)$. These are calculated using the BM25 algorithm, yielding two asymmetric scores. This approach recognizes that the impact of a morpheme can vary across different sentences, necessitating a nuanced evaluation of similarity. Then the harmonic mean is taken as the similarity score $Sim(D_1, D_2)$ between the two asymmetric scores:

$$Sim\,(D_1, D_2) = \frac{2 \times Sim\,(D_1 \rightarrow D_2) \times Sim\,(D_2 \rightarrow D_1)}{Sim\,(D_1 \rightarrow D_2) + Sim\,(D_2 \rightarrow D_1)}. \tag{1}$$

Due to the inheritance relationship, each tail API description set $\mathbb{D}$ includes both its description and the descriptions of its parents. However, the description of its parents cannot fully reflect the functionalities of a tail API. Therefore, we set up a semaphore DI as the weight to distinguish the impact of descriptions from different sources. After iterating

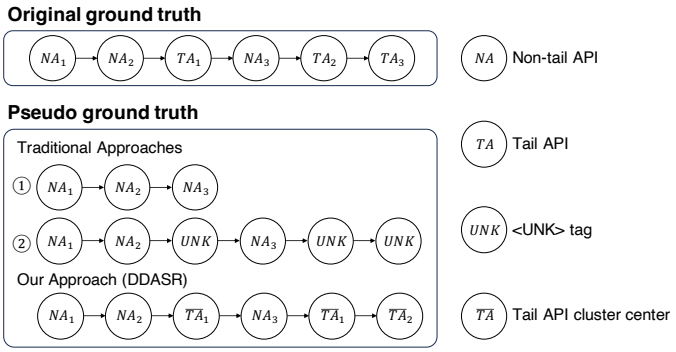


Fig. 4. The original ground truth and the pseudo ground truth of API sequence recommendation.

through all descriptions in $\mathbb{D}$, we take the maximum similarity as the description similarity $simD$ of two tail APIs.

$$SimD(TA_i, TA_j) = max\{DI \times Sim(D_{i,k}, D_{j,m}))|D_{i,k} \in \mathbb{D}_i \ and \ D_{j,m} \in \mathbb{D}_j\} \tag{2}$$

Similarity calculation of API names. We use the Levenshtein distance [35], a minimum edit distance algorithm, to calculate the similarity between the names of tail APIs. The Levenshtein distance is computed as the minimum number of three single-character edit operations, including deletion, insertion, and substitution (matching or mismatching), for converting one API name to another.

$$Lev_{N_i,N_j}(k,m) = \begin{cases} max(k,m), & min(k,m) = 0 \\ min\begin{pmatrix} Lev_{N_i,N_j}(k-1,m)+1, \\ Lev_{N_i,N_j}(k,m-1)+1, \\ Lev_{N_i,N_j}(k-1,m-1)+1_{N_i[k]\neq N_j[m]} \end{pmatrix}, & otherwise \end{cases}, \tag{3}$$

where N_i and N_j represent the names of tail APIs TA_i and TA_j, respectively. The indices k and m correspond to the positions within N_i and N_j, respectively. The function $Lev_{N_i,N_j}(k,m)$calculates the distance between the first k characters of N_i and the first m characters of N_j. In the minimum operation, the first term accounts for deletion implying the removal of a character from N_i to match N_j. The second term accounts for insertion, denoting the addition of a character to N_i. The final term represents the cost of substitution, which is determined by the indicator function $1_{N_i[k]\neq N_j[m]}$. The function yields 1 if $N_i[k] \neq N_j[m]$, indicating a mismatch, and 0 if they match.

The name similarity, $simN$, is calculated based on the Levenshtein distance:

$$simN(TA_i, TA_j) = 1 - \frac{Lev_{N_i,N_j}(|N_i|,|N_j|)}{max(|N_i|,|N_j|)}, \tag{4}$$

where $|N_i|$ and $|N_j|$ represent the length of N_i and N_j, respectively. $Lev_{N_i,N_j}(|N_i|,|N_j|)$ equals 0 when $N_i = N_j$.

Functional similarity matrix. Due to the different calculation methods of description similarity $SimD$ and name similarity $SimN$, they exhibit a significant difference in scale. Therefore, we employ *Min-Max Normalization* to standardize their scales. When the normalized similarity measures $SimD$ and $SimN$ exceed respective thresholds δ_1 and δ_2, the APIs in question are considered to have related descriptions or names. Otherwise, their similarity is deemed irrelevant and is effectively set to zero. The description is more important than the name for similarity calculation between APIs [32], so we set $\alpha = 0.8$ and $\beta = 0.2$, where α and β represent the weights of $simD$ and $simN$, respectively.

The final similarity score, Sim, is calculated by adding $simD$ and $simN$ after normalizing them with weights:

$$Sim(TA_i, TA_j) = \alpha \times max(\frac{simD(TA_i, TA_j)}{simD_{max} - simD_{min}} - \delta_1, 0) + \beta \times max(\frac{simN(TA_i, TA_j)}{simN_{max} - simN_{min}} - \delta_2, 0). \quad (5)$$

We calculate the similarity of all tail APIs in pairs to obtain an $n \times n$ symmetric similarity score matrix **SM**.

3.2.2 *Clustering and Replacing.*

To improve the representation of tail APIs, we build a pseudo ground truth by clustering the tail APIs based on the functional similarity matrix and replacing the tail APIs with cluster centers.

For a given query Q in the ground truth, the answer is a sequence that consists of a series of APIs, such as $< NA_1, NA_2, TA_1, NA_3, TA_2, TA_3 >$. Traditional API sequence recommendation approaches treat the sequence as $< NA_1, NA_2, NA_3 >$ or $< NA_1, NA_2, < UNK >, NA_3, < UNK >, < UNK >>$ since they delete tail APIs, i.e., TA_1, TA_2 and TA_3, or use *<UNK>* tags to replace them in the preprocessing phase as shown in Fig. 4. We hope to recommend tail APIs for developers. However, we cannot directly use the individual tail APIs as targets during training due to their insufficient occurrences. Thus, we utilize tail APIs after spectral clustering [53] to effectively represent the features of similar individual tail APIs.

We treat the similarity matrix **SM** of tail APIs as an adjacency matrix and use it to calculate a diagonal symmetric matrix with dimension $n \times n$, denoted as $\mathbf{D}^{iag}$:

$$\mathbf{D}^{iag}{}_{ij} = \begin{cases} \sum_k \mathbf{SM}_{ik}, & i = j,\ 0 \le k < n \\ 0, & i \neq j \end{cases}. \quad (6)$$

Based on Equation (6), we define a normalized symmetric Laplace matrix $\mathbf{L}_{rsym}$:

$$\mathbf{L}_{rsym} = \mathbf{D}^{iag^{-\frac{1}{2}}}(\mathbf{D}^{iag} - \mathbf{SM})\mathbf{D}^{iag^{\frac{1}{2}}}. \quad (7)$$

Algorithm 1 generates sample points based on tail APIs and their similarity matrix **SM** for clustering. We begin by computing $\mathbf{D}^{iag}$ and $\mathbf{L}_{rsym}$ (Lines 1-4). Then we calculate the eigenvalues of $\mathbf{L}_{rsym}$ and select the first c eigenvalues after sorting in ascending order. After calculating the eigenvectors of eigenvalues, we take them as column vectors to

Algorithm 1 Generate sample points for spectral clustering.

Input: **SM**: Similarity matrix of tail APIs. c: number of clusters.
Output: $\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$: Sample points.
Begin
1: /* Compute the diagonal matrix. */
2: $\mathbf{D}^{iag} \leftarrow diag(\sum_k \mathbf{SM}_{0k}, \sum_k \mathbf{SM}_{0k}, \ldots, \sum_k \mathbf{SM}_{0k})$
3: /*Compute the normalized Laplace matrix. */
4: $\mathbf{L}_{rsym} \leftarrow \mathbf{D}^{iag^{-1/2}}(\mathbf{D}^{iag} - \mathbf{SM})\mathbf{D}^{iag^{1/2}}$
5: /*Compute the first c eigenvectors $u_1, u_2, \ldots, u_c$ of $\mathbf{L}_{rsym}$. */
6: $\mathbf{U} \leftarrow [u_1, u_2, \ldots, u_c] \in \mathbf{R}^{n \times c}$
7: /*Compute matrix **T** from matrix **U** by individually normalizing each row with L_2-norm. */
8: $\mathbf{T} \leftarrow [t_{ij} = u_{ij} / \sqrt{\sum_{m=1}^{n} u_{im}^2}] \in \mathbf{R}^{n \times c}$
9: /*Define $y_i \in \mathbf{R}^c$ as the vector associated with the i-th row of matrix **T**. */
10: $\mathbf{Y} \leftarrow \{y_i = row_i(\mathbf{T})\}_i^n$
11: **return** **Y**

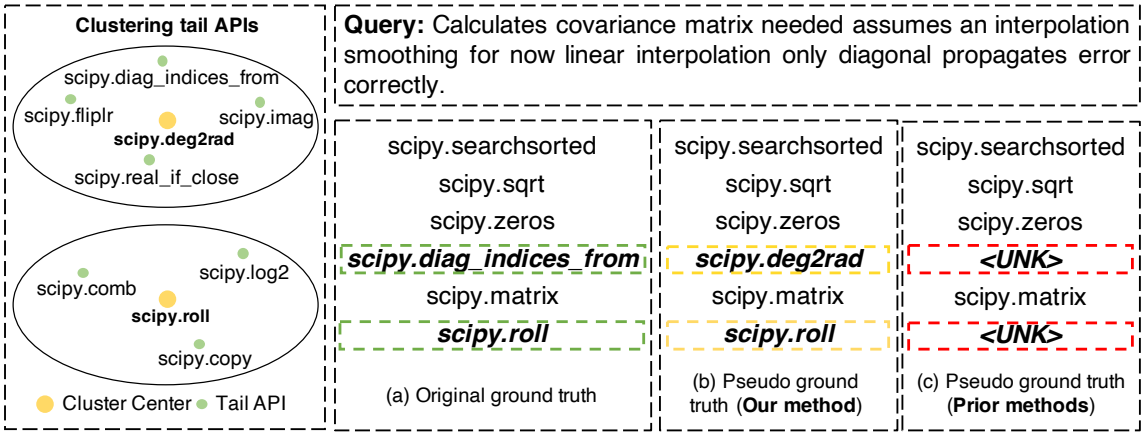


Fig. 5. An example of building the pseudo ground truth. (a) The original ground truth, (b) the pseudo ground truth built by clustering tail APIs and replacing them with cluster centers (as done in our method), and (c) the pseudo ground truth built by replacing tail APIs with *<UNK>* for the query (as done in prior methods).

form the matrix **U** (Lines 5-6). Finally, the vectors in each row of **U** are converted to unit vectors to form new sample points **Y** (Lines 7-10).

After that, we use the K-means algorithm to assign tail APIs with similar functionality into a cluster $\overline{\mathbf{TA}} = \{\overline{\mathbf{TA}}_1, \overline{\mathbf{TA}}_2, \ldots, \overline{\mathbf{TA}}_j, \ldots \overline{\mathbf{TA}}_c\}$ based on sample points Y, where c represents the number of cluster centers. For the cluster $\overline{\mathbf{TA}}_j$, $\overline{TA}_j$ is the cluster center. Recommending cluster centers can expose some of the hidden tail APIs and increase the diversity of recommendation results, and can also inspire developers due to the functional similarity of the exposed and hidden tail APIs in the clusters. Therefore, we substitute tail APIs with cluster centers. For an API sequence represented as $< NA_1, NA_2, TA_1, NA_3, TA_2, TA_3 >$, we restructure it to $< NA_1, NA_2, \overline{TA}_1, NA_3, \overline{TA}_1, \overline{TA}_2 >$ to build a pseudo ground truth, as illustrated in Fig. 4. This restructuring is based on the categorization of TA_1 and TA_2 as belonging to the same cluster $\overline{\mathbf{TA}}_1$, while TA_3 is classified under a different cluster $\overline{\mathbf{TA}}_2$. Fig. 5 shows an example of building the pseudo ground truth. There are two clusters of tail APIs as shown in Fig. 5. For the query, Fig. 5(a) is the original ground truth, Fig. 5(b) is the pseudo ground truth built by clustering tail APIs and relocating with the cluster centers, and Fig. 5(c) is the pseudo ground truth constructed by replacing tail APIs with *<UNK>* tags. Moreover, the number of occurrences of each cluster is the sum of the number of occurrences of tail APIs belonging to the cluster. However, the clusters serve as substitutes for the tail APIs in the training process. Each tail API cluster should not appear more frequently than the non-tail APIs. Thus, we set the number of clusters c such that the average number of tail APIs in clusters approximates that of non-tail APIs, and ensure that the item count in each cluster does not substantially exceed that of non-tail APIs.

3.3 Model Training

3.3.1 *Sequence-to-Sequence Model.*

DDASR applies Seq2Seq, an encoder-decoder model, to generate API sequences. For a given query, the encoder can produce a contextualized embedding vector, which is used by the decoder to produce an API sequence. The LLM encoder-decoder architecture is used in DDASR, with LLMs serving as the encoder to capture developer requirements, and a six-layer Transformer acting as the decoder.

3.3.2 Learning-to-Rank Loss Function.

To achieve a balance between accuracy and diversity, we adopt ListMLE [59], a listwise learning-to-rank method, as our loss function. By utilizing ListMLE to model the rankings of all API sequences generated, we learn the order of ranking to balance the effect of tail API clusters on diversity by improving the prediction accuracy of the rankings, thus allowing the model to increase diversity without sacrificing accuracy. For a given query Q and corresponding API sequence A in the pseudo ground truth, we define a sequence probability distribution based on Plackett-Luce [59] for the recommendation results as follows:

$$P(\mathbf{A}|s) = \prod_{i=1}^{m} \frac{\varphi(s_{A^{-1}(i)})}{\sum_{u=i}^{m} \varphi(s_{A^{-1}(u)})}, \tag{8}$$

where $A^{-1}(i)$ represents the APIs sorted to the i-th positions in sequence $\mathbf{A} =< A_1, A_2, \cdots, A_m >$. $s_{A^{-1}(i)}$ is a ranking score of item A_i from ranking scores vector $s = f(x)$ and $\varphi()$ is an exponential mapping function, namely $\varphi(x) = exp(x)$. The better the fit between $\mathbf{A}$ and s, the higher the probability value $P(\mathbf{A}|s)$.

We define our loss function L as the inverse of log-likelihood:

$$L = -logP(\mathbf{A}|f(x)), \tag{9}$$

where L is a convex function so that we can optimize it directly with gradient descent.

Finally, DDASR produces top-k API sequences for each given query by using Beam Search [31], a heuristic search strategy based on the average loss.

4 EVALUATION

In this section, we evaluate the proposed DDASR approach. We will study the following three research questions (RQs):

- RQ1: How accurate is DDASR in comparison with the state-of-the-art approaches?
- RQ2: Does DDASR improve diversity while maintaining accuracy?
- RQ3: Can DDASR help programmers address tasks more effectively?

4.1 Experiment

All our experiments were performed on a Supermicro GPU server with Intel(R) Xeon(R) E5-2640 v4 x86_64, 2.4GHz, 2 GPUs (NVIDIA Tesla V100 16GB), and the CentOS 7.5 Linux operating system. The machine learning models were implemented with the PyTorch library.

4.2 Data

We utilize two open-source datasets: one for the Java programming language [19] and the other for the Python programming language [40]. The Java dataset constructed by Gu et al. [19] encompasses seven million natural language annotations, considered as queries, alongside API sequences. These are extracted by mining Java projects on GitHub that have garnered more than one star. However, in the original Java dataset, we find about six million duplicate records and many errors in the API sequences, as illustrated in Table 1. We eliminate the duplicate records. Subsequently, our examination revealed a total of 25,862 records containing mismatched symbols, with the predominant issue being a disparity in the count of left and right parentheses. To address this imbalance, we primarily rectified the records by adding parentheses where necessary to achieve symmetry. In addition, we find 3,552 records whose class or method names mismatch the official API documentation. We revise them manually by consulting the API documentation.

Table 1. Five data cleaning steps on the original Java dataset with corresponding scenarios, processing methods, and examples

Type	Scenarios	Processing Method	Example	Example After Processing
Duplicate pairs	Duplicate records for the same *query-APIseq* pairs in the dataset.	Delete duplicate records and keep only one.	-	-
Class name errors	Class name is inconsisten with the official API documentation.	Change to standard class name according to the official API documentation.	CDRInputStream_1_0.<init>	CDRInputStream.<init>
Method name errors	Method name is inconsistent with the official API documentation.	For those caused by spelling errors, make the necessary corrections. For methods name that do not exist, remove the entire API.	ConcurrentLinkedDeque.succ	-
	Not all actual arguments have been completely removed.	Remove the actual arguments from the method name.	StringBuilder.append("time")	StringBuilder.append
Mismatched symbols	The number of left and right parentheses is unequal.	Complete the mismatched parentheses.	ArrayList<.append	ArrayList<>.append

Table 2. Summary of the datasets

Datasets		# Pairs	# Pairs Containing Tail APIs	# Non-tail APIs	# Tail APIs
Java dataset	Training set	7,519,907	63,239	46,245	79,797
	Test set	9,975	175	12,895	227
Diverse Java dataset	Training set	752,919	62,654	46,180	78,521
	Test set	7,600	760	12,198	1,555
Python dataset	Training set	835,069	3,914	4,949	3,534
	Test set	10,000	131	1,914	153

Finally, we obtain a repository in Java, including about 760k *query-APIseq* pairs and 120k APIs. Additionally, we observe a phenomenon that the test set of the original dataset contains very few *query-APIseq* pairs of tail APIs. To evaluate the diversity in API sequence recommendation, we build a diverse Java dataset with an equal distribution of pairs containing tail APIs in the training and test sets by stratified random sampling. The Python dataset constructed by Martin and Guo [40] consists of about 855k records by collecting Python projects with at least five stars. After a comprehensive investigation, we do not find the issue shown in Table 1. Moreover, the Python dataset contains a balanced record of tail APIs in both the test and training sets and can be used to evaluate diversity without constructing a separate diverse dataset. Finally, we obtain the Python dataset which contains around 855k *query-APIseq* pairs and 8.5k APIs. Table 2 shows the summary of the datasets.

For the descriptions of tail APIs, we mine the API documentation based on the class names of long-tail APIs. When experimenting on Java, we download the Java SE 8 API documentation[2] and third-party documentation, such as Spring

[2] http://www.oracle.com/technetwork/java/javase/

Boot[3], JUnit[4], and Log4j[5]. When experimenting on Python, we crawl the official documentations of Python 3[6], Numpy[7], SciPy[8], PyTorch[9], and Pandas[10] to mine the descriptions.

4.3 Baselines

We choose four state-of-the-art approaches, DeepAPI [19], BIKER [24], CodeBERT [40], and CodeTrans [12], as well as DDASR, a variant of our approach, as competing models. In particular, BIKER is an information retrieval-based approach, while the others are generative approaches based on deep learning.

- **DeepAPI**[11] [19] adapts an RNN encoder-decoder model to encode a query into a fixed-length context vector and generate API sequences based on the context vector.
- **BIKER**[12] [24] extracts API-related posts of Stack Overflow, treating the titles of the questions as search queries and the APIs mentioned in the accepted answers as the canonical solutions. The tool mainly calculates the similarity of questions and recommends the APIs of similar questions. To adapt BIKER for API sequence recommendation, we make the necessary modifications to its open-source code.
- **CodeBERT**[13] is reproduced by Martin and Guo [40] to generate API sequences by fine-tuning CodeBERT [14], a pre-trained model, on the Java and Python datasets.
- **CodeTrans**[14] [12] combines encoder-decoder and Transformer models and explores different training strategies to recommend API sequences. In addition, CodeTrans provides a pre-trained model for the task of API sequence generation on the original Java dataset, which we utilize directly for experimental comparison.
- **DASR** is a variant of our approach, which can be analogous to the word embedding and character embedding of APIs, to evaluate the performance of the results when a complete API and API fragments are input-feeding.

4.4 Model Selection

We consider three types of encoder-decoder models in our experiment.

- **RNN Encoder-Decoder Model.** We choose the model because it is used to to generate API sequences by Gu et al. [19]. A standard Bidirectional Long Short Term Memory (Bi-LSTM) is employed in the encoder, and the decoder is a Gate Recurrent Unit (GRU) network.
- **Transformer Encoder-Decoder Model.** The encoder and decoder consist of six layers of Transformer encoder and decoder components. CodeTrans [12] also employs a similar architecture to accomplish tasks.
- **LLM Encoder-Decoder Model.** LLMs currently demonstrate outstanding performance in various natural language processing tasks. They are categorized into three architectures: encoder-only, decoder-only, and encoder-decoder. In our approach, we do not consider decoder-only models because our model combines API tokens into complete APIs that are not present in the vocabulary of LLMs. For the encoder-only and

[3] https://docs.spring.io/spring-boot/docs/2.6.12/api/
[4] https://junit.org/junit4/javadoc/latest/
[5] https://logging.apache.org/log4j/2.x/log4j-api/apidocs/
[6] https://docs.python.org/3.8/
[7] https://numpy.org/doc/stable/reference/
[8] https://docs.scipy.org/doc/scipy/reference/
[9] https://pytorch.org/docs/stable/
[10] https://pandas.pydata.org/docs/reference/
[11] https://github.com/huxd/deepAPI
[12] https://github.com/tkdsheep/BIKER-ASE2018
[13] https://github.com/hapsby/deepAPIRevisited
[14] https://github.com/agemagician/CodeTrans

encoder-decoder LLMs, we only utilize their encoder component as the encoder for our approach. We select five widely available LLMs for code in the HuggingFace API[15], including CodeBERT[16], GraphCodeBERT[17], PLBART[18], CodeT5[19], and UniXcoder[20]. CodeBERT [14] is an encoder-only model pre-trained on bimodal data sourced from CodeSearchNet [25]. GraphCodeBERT [21] is also an encoder-only model that incorporates syntax information from code. The other three LLMs are encoder-decoder models. PLBART [3] is pre-trained on a wide range of Java and Python function collections and related natural language text using denoising auto-encoders. CodeT5 [54] can better leverage the code semantics conveyed from the developer-assigned identifiers. UniXcoder [20] comprehensively utilizes the code structure information provided by the Abstract Syntax Tree (AST). The decoder consists of six layers of Transformer decoder.

We use the following hyper-parameter settings. For the RNN encoder-decoder, we set the number of hidden layers, number of hidden units, and dimension of word embedding to be 3, 1,000, and 512, respectively. For the Transformer encoder-decoder, both the encoder and decoder have six layers and the word embedding dimension is 512. For the LMM encoder-decoder, the hidden dimensions for the LMMs used as encoder are 768, and the decoder has six layers. The AdamW algorithm is used to optimize model parameters with the learning rate $\alpha = 2e - 5$. Moreover, we adopt the optimal settings of baselines reported in the literature for fair comparison.

4.5 Metrics

- **BLEU** (Bilingual Evaluation Understudy) is used to evaluate how close the candidate API sequences generated by the model are to the reference API sequence in the ground truth.
- **MAP** (Mean Average Precision) is the mean of the average precision (AP) score for each query.
- **NDCG** (Normalized Discounted Cumulative Gain) evaluates the ranking quality of candidate API sequences.
- **Coverage** [27] is used to assess the diversity of API sequence recommender systems. It is defined as the proportion of all candidate API sequences exposed to APIs in the vocabulary.

BLEU, MAP, and NDCG are metrics for evaluating the accuracy. However, previous work has typically used the pseudo ground truth as a reference for evaluating, which we refer to as the falsified accuracy. This is because, in the pseudo ground truth, tail APIs are replaced with *<UNK>* tags or cluster centers, leading to evaluation bias. We denote the accuracy metrics evaluated using the pseudo ground truth as BLEU_P, MAP_P, and NDCG_P. To restore the veritable accuracy, we also utilize the original ground truth as a reference, and in this case, the accuracy metrics are labeled as BLEU_O, MAP_O, and NDCG_O. Moreover, the smallest unit of evaluation in our experiment is a complete API to avoid recommending wrong APIs for developers.

4.6 Experimental Results

4.6.1 *RQ1. How accurate is DDASR in comparison with the state-of-the-art approaches?*

Motivation. A crucial evaluating factor for API sequence recommendation is accuracy. In DDASR, we have incorporated a treatment for tail APIs. We evaluate the accuracy of DDASR concerning baselines on the original Java dataset.

[15] https://huggingface.co
[16] https://huggingface.co/microsoft/codebert-base
[17] https://huggingface.co/microsoft/graphcodebert-base
[18] https://huggingface.co/uclanlp/plbart-base
[19] https://huggingface.co/Salesforce/codet5-base
[20] https://huggingface.co/microsoft/unixcoder-base

Table 3. Evaluation results in accuracy on the original Java dataset

	Models	BLEU_P			MAP_P			NDCG_P			p
		top-1	top-5	top-10	top-1	top-5	top-10	top-1	top-5	top-10	
Baselines	DeepAPI	0.2818	0.3659	0.3738	0.3672	0.4308	0.4374	0.3672	0.3572	0.3545	-
	BIKER	0.0000	0.0000	0.0000	0.2892	0.3471	0.4512	0.0700	0.0612	0.0588	-
	CodeBERT	0.4024	0.4677	0.4815	0.4774	0.5463	0.5618	0.4774	0.3753	0.3522	-
	CodeTrans	0.4178	0.4708	0.5006	0.5049	0.5750	0.6115	0.5049	0.3655	0.3420	-
DDASR	DDASR(RNN)	0.6510	0.7601	0.7825	0.7860	0.8521	0.8684	0.7860	0.7489	0.7316	***
	DDASR(Transformer)	0.6575	0.7685	0.7895	0.7970	0.8632	0.8788	0.7970	0.7903	0.7876	***
	DDASR(CodeBERT)	**0.6682**	0.7707	0.8205	**0.8158**	0.8737	0.8971	**0.8158**	**0.8123**	**0.8104**	***
	DDASR(GraphCodeBERT)	0.6628	0.7742	0.8218	0.8135	**0.8784**	**0.8998**	0.8135	0.8109	0.8093	***
	DDASR(PLBART)	0.6649	0.7703	0.8191	0.8129	0.8751	0.8978	0.8129	0.8080	0.8062	***
	DDASR(CodeT5)	0.6484	0.7461	0.8027	0.8026	0.8604	0.8871	0.8026	0.8000	0.8000	***
	DDASR(UniXcoder)	0.6665	**0.7776**	**0.8246**	0.8137	0.8765	0.8993	0.8137	0.8092	0.8070	***

Note: "***" represents a highly significant statistical difference (p<0.001) and "-" represents that there is no value in this cell.

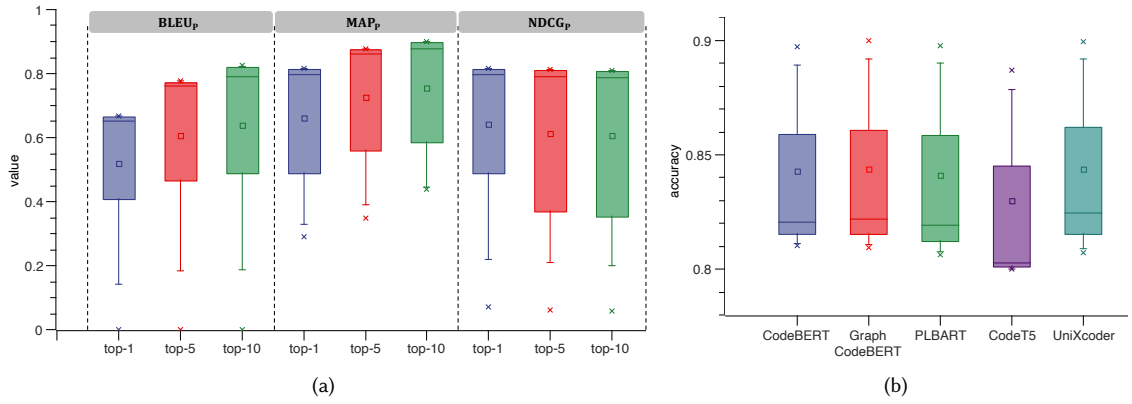


Fig. 6. Accuracy comparative analysis of DDASR on the original Java dataset. (a) Comparison of top-k results under different accuracy metrics, and (b) comparison of DDASR with different LLMs.

Approach. We first train DDASR on different architectures and DeepAPI with the aforementioned settings on the original Java dataset. For CodeBERT and CodeTrans, we directly utilize their publicly available models to generate API sequences. For BIKER, an information retrieval-based approach, we execute the open-source code after necessary modifications.

We assess the performance using accuracy metrics, specifically BLEU_P, MAP_P, and NDCG_P, across the top-1, top-5, and top-10 recommendation results. We have opted not to employ conventional accuracy metrics for evaluation since in the test set, there is a scarcity of tail APIs, and the constructed pseudo ground truth has little impact on the veritable accuracy of the evaluation results.

We perform *T-test*, a statistical test, which tests for differences in means between two groups are used to find if differences in reported performance are statistically significant. We first examine the significant differences between

DDASR with different architectures and the best-performing baseline. We utilize *T-test* to test the statistical differences among top-1, top-5, and top-10 results. In addition, the significant differences among DDASR with different LLMs when applying the LLM encoder-decoder architecture are also assessed.

Results. Table 3 shows the evaluation results on the original Java dataset. The bold numbers in the table represent the optimal outcomes. The *p*-value is the result of the *T-test* between DDASR and the best-performing baseline. Fig. 6 shows the comparison of top-*k* results and the comparison of DDASR using different LLMs.

(1) Compared to the baselines, DDASR demonstrates significant accuracy. According to Table 3, DDASR significantly improves all metrics ($p<0.05$), regardless of which architecture it uses. The accuracy of top-1, top-5, and top-10 results is more than 55.19%, 58.47%, and 56.31% on $BLEU_P$, 55.67%, 48.19%, and 42.01% on MAP_P, as well as 55.67%, 99.55%, and 106.38% on $NDCG_P$, respectively, which shows DDASR can provide correct API sequences in most cases. In the deep learning-based baselines, APIs are broken down into word or symbol fragments, whereas in our metric, any sign or letter error in an API is deemed an error API. It demonstrates that feeding vector representations into the neural network model with the complete APIs instead of the segmented fragments can efficiently avoid mistakes in the components of the APIs. The $BLEU_P$ of BIKER is about 0, indicating that the API sequences recommended by BIKER do not retain the correct call relationship. In addition, CodeBERT performs best as the encoder of DDASR in terms of accuracy in the top-1 recommendation results. In the top-5 and top-10 recommendation results, UniXcoder excels in terms of $BLEU_P$ metric performance, and GraphCodeBERT, on the other hand, demonstrates the best performance in terms of MAP_P metric, and when it comes to $NDCG_P$ metrics, CodeBERT outperforms other encoders. From Fig. 6(b), it can be observed that DDASR using GraphCodeBERT demonstrates the best overall performance.

(2) An increase in the number of recommended results tends to result in improved accuracy. As the number of recommendation results rises, the $BLEU_P$ and MAP_P increase. It indicates that developers are more likely to discover the optimal solution when more results are offered. Nevertheless, the growth is not significant ($p>0.05$). In addition, $NDCG_P$ decreases as the number of recommendation results increases. It verifies that the first appearance of correct APIs is concentrated in the position in front of the recommendation results.

(3) The performance gap of DDASR with the LLM encoder-decoder architecture is not very significant when applying different LLMs ($p>0.05$). In the top-10 recommended results, the differences of DDASR with different LLMs in $BLEU_P$, MAP_P, and $NDCG_P$ are only 0.67%, 1.43%, and 1.30%, respectively.

Answer to RQ1: When using alternative encoder-decoder architectures on the original Java dataset, DDASR consistently outperforms the baseline models in terms of accuracy. Furthermore, the performance of DDASR with different LLM encoder-decoder models is not pronounced.

4.6.2 *RQ2. Does DDASR improve diversity while maintaining accuracy?*

Motivation. Diversity is another essential factor in API sequence recommendation, which is disregarded by previous approaches. In RQ2, we look at how well our DDASR performs in terms of diversity. Furthermore, accuracy and diversity are frequently incompatible objectives. We also investigate how DDASR affects accuracy when improving diversity.

Approach. To answer this question, we conduct comparative experiments at the top-10 recommendation results on the diverse Java dataset and the Python dataset. We train DDASR and DASR on different architectures as well as deep learning-based baselines with the aforementioned settings on the diverse Java dataset and the Python dataset, respectively. Specifically, when working on the Python dataset, we directly utilize the publicly available model of CodeBERT to generate results. The execution of BIKER is the same as RQ1.

Table 4. Evaluation results in accuracy and diversity at top-10 recommendation results on the diverse Java dataset

Models		$BLEU_P$	$BLEU_O$	MAP_P	MAP_O	$NDCG_P$	$NDCG_O$	Coverage	p
Baselines	DeepAPI	0.0943	0.0941	0.1815	0.1804	0.1506	0.1401	0.0329	-
	BIKER	0.0000	0.0000	0.0554	0.0554	0.0039	0.0039	0.0319	-
	CodeBERT	0.0464	0.0402	0.0909	0.0896	0.0347	0.0309	0.0544	-
	CodeTrans	0.2477	0.2389	0.3375	0.3246	0.1324	0.1276	0.0744	-
DASR	DASR(RNN)	0.2926	0.2865	0.4557	0.4442	0.4243	0.4168	0.0400	***
	DASR(Transformer)	0.4124	0.4027	0.4865	0.4796	0.4268	0.4127	0.0406	***
	DASR(CodeBERT)	**0.5260**	**0.5086**	0.6231	0.6136	**0.4752**	**0.4621**	0.0430	***
	DASR(GraphCodeBERT)	0.4711	0.4609	0.5885	0.5801	0.4119	0.4093	0.0418	***
	DASR(PLBART)	0.4769	0.4688	0.5876	0.5796	0.4249	0.4117	0.0438	***
	DASR(CodeT5)	0.4992	0.4911	0.6007	0.5943	0.4521	0.4474	0.0446	***
	DASR(UniXcoder)	0.4851	0.4824	0.6056	0.5994	0.4208	0.4165	0.0428	***
DDASR	DDASR(RNN)	0.3221	0.3196	0.4493	0.4438	0.4526	0.4435	0.1064	***
	DDASR(Transformer)	0.4002	0.3971	0.4968	0.4883	0.4344	0.4263	0.0758	***
	DDASR(CodeBERT)	0.5159	0.5077	**0.6365**	**0.6255**	0.4629	0.4554	**0.1085**	***
	DDASR(GraphCodeBERT)	0.4629	0.4581	0.5845	0.5769	0.4115	0.4073	0.1038	***
	DDASR(PLBART)	0.4550	0.4516	0.5832	0.5770	0.4067	0.4037	0.0996	***
	DDASR(CodeT5)	0.4880	0.4826	0.6100	0.6012	0.4404	0.4343	0.0978	***
	DDASR(UniXcoder)	0.4759	0.4702	0.6018	0.5927	0.4199	0.4154	0.1003	***

Note: "***" represents a highly significant statistical difference ($p<0.001$) and "-" represents that there is no value in this cell.

We calculate the accuracy metrics of $BLEU_P$, MAP_P, and $NDCG_P$, the veritable accuracy metrics of $BLEU_O$, MAP_O, and $NDCG_O$, as well as the diversity metric of coverage on the top-10 results of all models. Moreover, we conduct significance analysis for performance differences between DASR and the best-performing baseline, between DDASR and the best-performing baseline, among DASR with different LLM encoder-decoder architectures, among DDASR with different LLM encoder-decoder architectures, between DASR and DDASR on coverage, and between DASR and DDASR on accuracy.

Results. Table 4 shows the experimental results on the diverse Java dataset. Fig. 7 shows the change of accuracy metrics in DASR and DDASR with epoch during the training phase. Among them, DASR and DDASR employ seven encoder-decoder models. The performance results on the Python dataset are shown in Table 5. The p-value in Table 4 and Table 5 is the result of the *T-test* between DASR and the best-performing baseline, as well as between DDASR and the best-performing baseline. Fig. 8 shows the comparison of DASR using different LLMs and the comparison of DDASR using different LLMs on the diverse Java dataset. Fig. 9 shows the comparison of DASR and DDASR on accuracy and the comparison of DASR and DDASR on coverage. Fig. 10 shows the comparison of DASR and DDASR with different LLMs on accuracy on the diverse Java dataset. Fig. 11, Fig. 12, and Fig. 13 show the comparison on the Python dataset.

(1) Both DASR and DDASR exhibit significant accuracy gains over baselines ($p<0.05$), and the difference between DASR and DDASR is not particularly noticeable ($p>0.05$) on the diverse Java dataset. As can be seen from Table 4, accuracy metrics for both DASR and DDASR have significantly increased in comparison to the baselines ($p<0.05$), which illustrates that word embedding of the complete API outperforms character embedding of API fragments. Regardless of

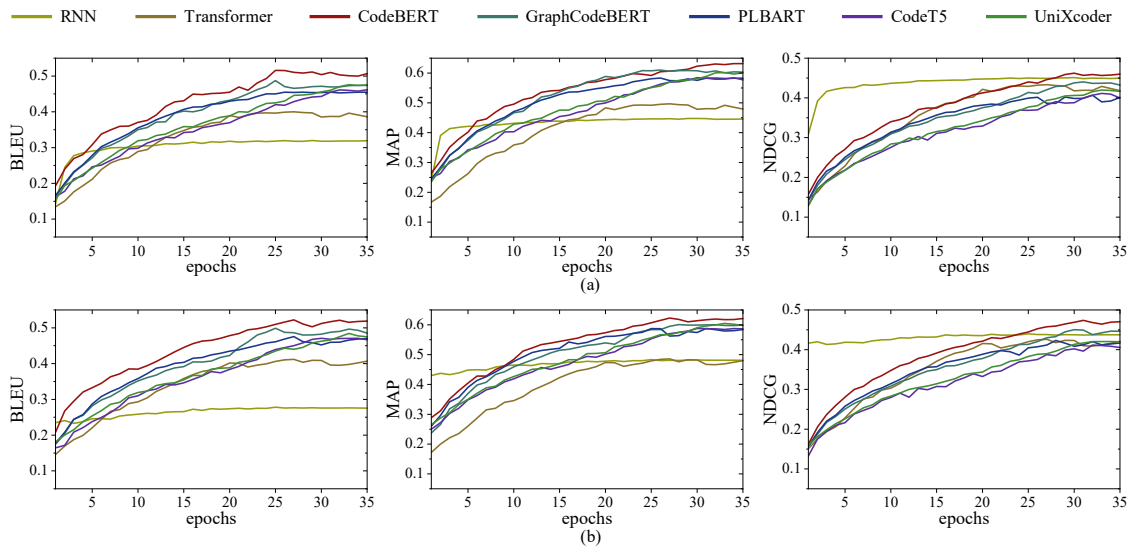


Fig. 7. The comparison of BLEU, MAP, and NDCG for each epoch on the diverse Java dataset. (a) The performance of DDASR, and (b) the performance of DASR.

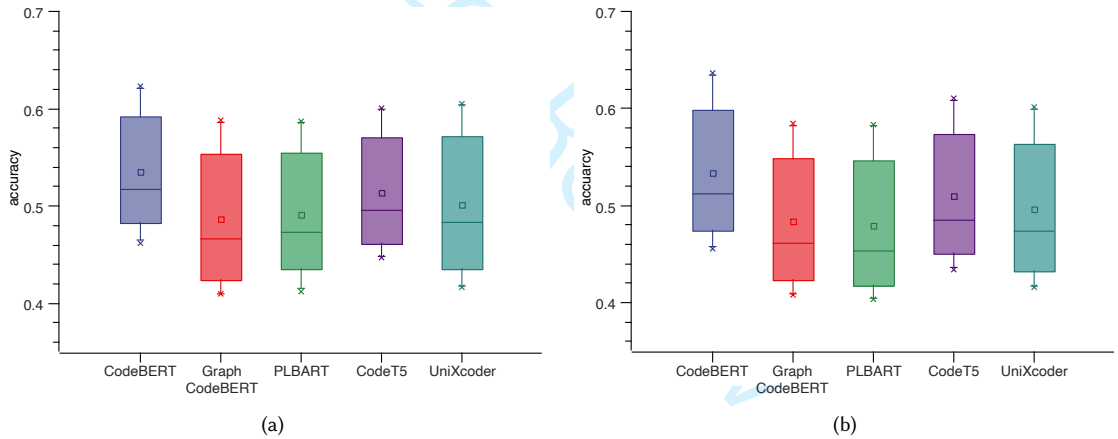


Fig. 8. Accuracy comparative analysis of DASR and DDASR applying different LLMs on the diverse Java dataset. (a) Comparison of DASR with different LLMs applied, and (b) comparison of DDASR with different LLMs.

which LLMs DDASR uses, DDASR exceeds baselines with the best performance by more than 83.69%, 89.03%, 77.73%, 101.54%, 109.77%, and 188.15% on all accuracy metrics. From Fig. 7, we can observe that DASR and DDASR using CodeBERT as the encoder consistently outperform other models. As shown in Fig. 9(a), the difference is not particularly apparent ($p>0.05$) between DASR and DDASR. Moreover, for DDASR and DASR, which use the same LLMs, DDASR has not shown a significant decrease in accuracy, and in some cases, it even outperforms DASR as shown in Fig. 10. It

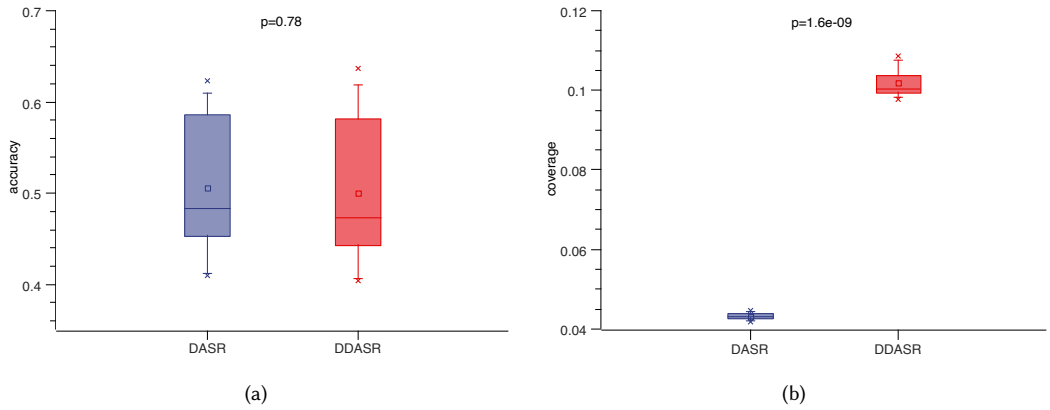


Fig. 9. Comparative analysis between DASR and DDASR on the diverse Java dataset. (a) Comparison on accuracy, and (b) comparison on coverage.

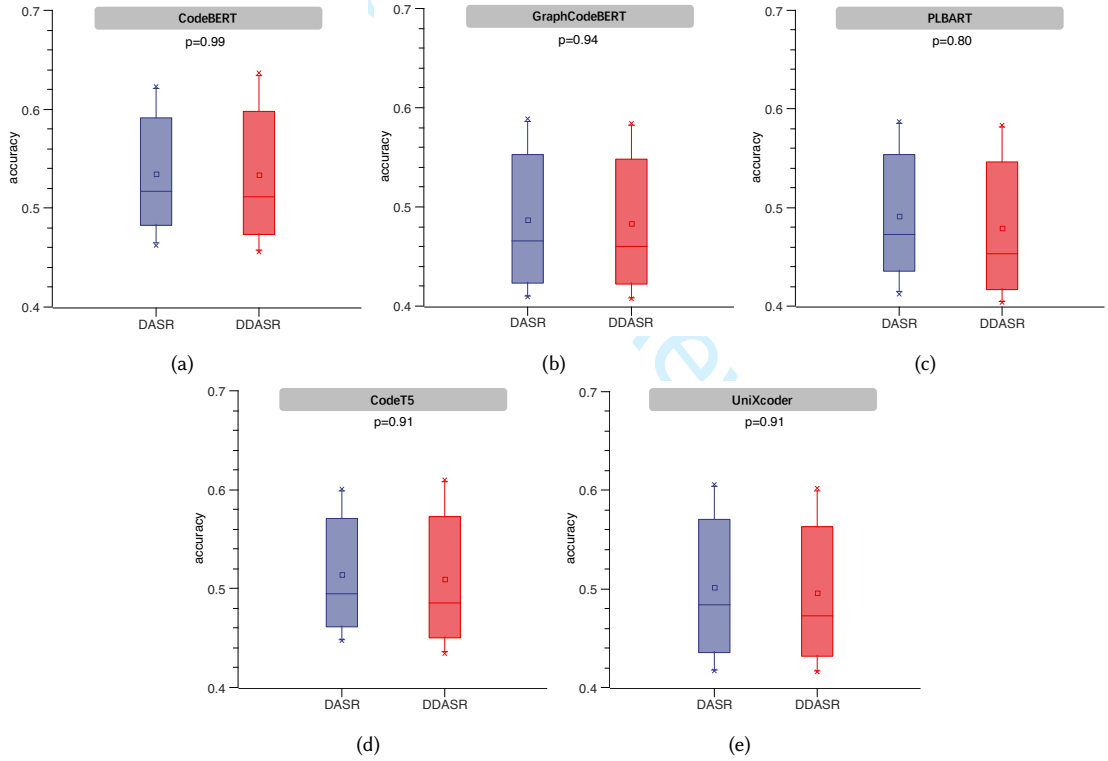


Fig. 10. Accuracy comparative analysis between DASR and DDASR with different LLMs on the diverse Java dataset. (a) Comparison with CodeBERT, (b) comparison with GraphCodeBERT, (c) comparison with PLBART, (d) comparison with CodeT5, and (e) comparison with UniXcoder.

Table 5. The performance in accuracy and diversity at top-10 recommendation results on the Python dataset

Models		$BLEU_P$	$BLEU_O$	MAP_P	MAP_O	$NDCG_P$	$NDCG_O$	Coverage	p
Baselines	DeepAPI	0.4386	0.4227	0.5012	0.4835	0.3561	0.3342	0.1851	-
	BIKER	0.1865	0.1865	0.4037	0.4037	0.3158	0.3158	0.1164	-
	CodeBERT	0.4099	0.3896	0.5249	0.5041	0.1836	0.1702	0.2080	-
	CodeTrans	0.3998	0.3854	0.5176	0.4988	0.1901	0.1819	0.1924	-
DASR	DASR(RNN)	0.4843	0.4754	0.5440	0.5229	0.3767	0.3584	0.1475	ns
	DASR(Transformer)	0.4982	0.4799	0.5845	0.5770	0.2752	0.2698	0.2320	ns
	DASR(CodeBERT)	**0.5432**	**0.5240**	**0.6117**	**0.6032**	0.3660	0.3587	0.1954	ns
	DASR(GraphCodeBERT)	0.5058	0.4896	0.5781	0.5704	0.3206	0.3146	0.1786	ns
	DASR(PLBART)	0.5098	0.4927	0.5860	0.5810	0.3234	0.3150	0.1784	ns
	DASR(CodeT5)	0.5389	0.5228	0.6090	0.6017	0.3633	0.3572	0.1736	ns
	DASR(UniXcoder)	0.4885	0.4719	0.5682	0.5610	0.2881	0.2824	0.1946	ns
DDASR	DDASR(RNN)	0.4560	0.4507	0.5219	0.5134	0.3211	0.3117	0.2232	ns
	DDASR(Transformer)	0.4292	0.4128	0.5108	0.5039	0.2546	0.2405	**0.2378**	ns
	DDASR(CodeBERT)	0.4858	0.4679	0.5672	0.5592	0.3677	0.3605	0.2247	ns
	DDASR(GraphCodeBERT)	0.4928	0.4768	0.5726	0.5620	**0.3879**	**0.3809**	0.2103	ns
	DDASR(PLBART)	0.4869	0.4738	0.5682	0.5619	0.3609	0.3558	0.2216	ns
	DDASR(CodeT5)	0.4544	0.4453	0.5295	0.5290	0.3603	0.3543	0.2256	ns
	DDASR(UniXcoder)	0.4984	0.4828	0.5776	0.5668	0.3843	0.3758	0.2177	ns

Note: "ns" represents no significance ($p>0.05$) and "-" represents that there is no value in this cell.

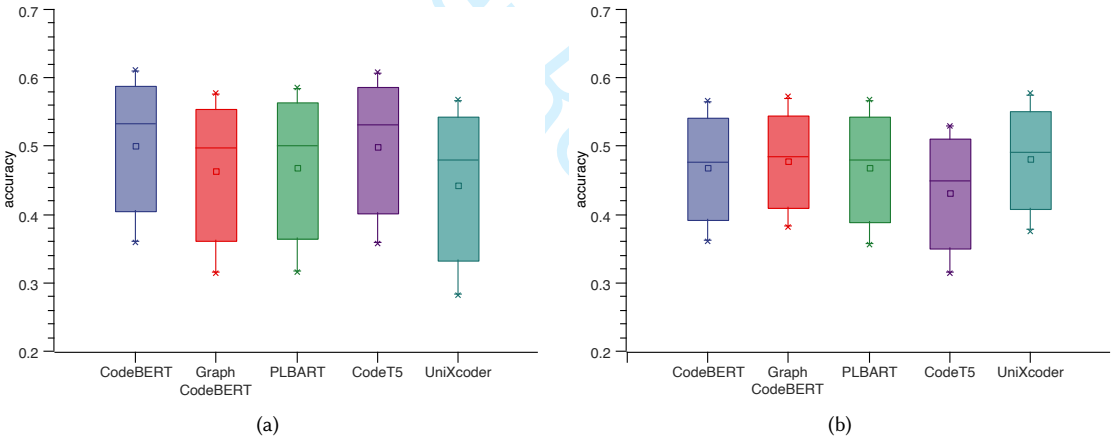


Fig. 11. Accuracy comparative analysis of DASR and DDASR applying different LLMs on the Python dataset. (a) comparison of DASR with different LLMs, and (b) comparison of DDASR with different LLMs.

indicates that the treatment of tail APIs in DDASR does not cause a significant drop in the accuracy of recommendation results.

(2) Both DASR and DDASR with LLMs perform better than baselines in terms of accuracy, and the differences are not particularly noticeable ($p>0.05$) on the Python dataset. Regardless of which LLMs DDASR uses, DDASR exceeds baselines with the best performance by more than 3.60%, 5.34%, 4.94%, 1.20%, 2.47%, and 6.01% on all accuracy metrics,

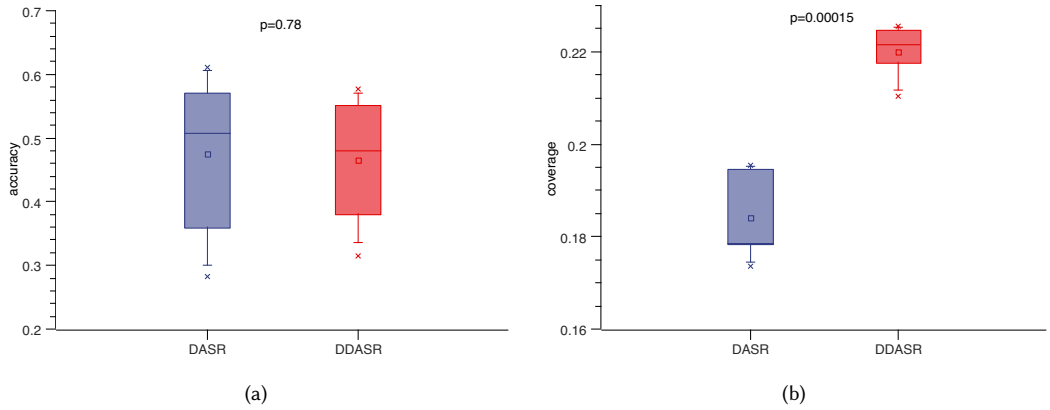


Fig. 12. Comparative analysis between DASR and DDASR on the Python dataset. (a) Comparison on accuracy, and (b) comparison on coverage.

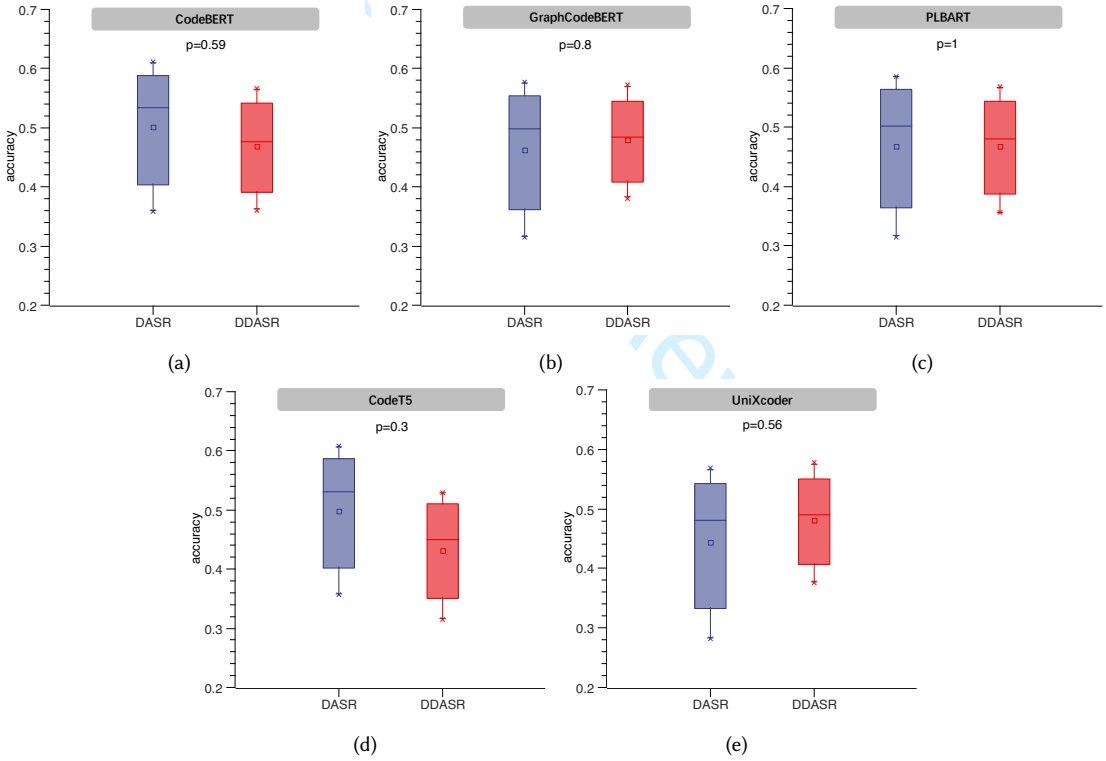


Fig. 13. Accuracy comparative analysis between DASR and DDASR with different LLMs on the Python dataset. (a) Comparison with CodeBERT, (b) comparison with GraphCodeBERT, (c) comparison with PLBART, (d) comparison with CodeT5, and (e) comparison with UniXcoder.

although the difference is not significant ($p>0.05$). We speculate it is due to the more straightforward composition of Python APIs. For example, "*scipy.special.digamma*" is a typical API in the Python dataset. There are essentially no symbols like "<" and ">" in API, and no nested usage patterns as in the Java dataset, making the effect of merging APIs less obvious than in the Java dataset. Furthermore, it can be observed from Fig. 12(a) and Fig. 13 that, on the Python dataset, the accuracy difference between DASR and DDASR is not significant.

(3) On both the diverse Java dataset and the Python dataset, the restored veritable accuracy is lower than the original accuracy, but DASR and DDASR still perform better than baselines. Compared to BLEU_P, MAP_P, and NDCG_P, BLEU_O, MAP_O, and NDCG_O all show a decrease. The reduction for DASR and deep learning-based baselines, however, is frequently more than for DDASR. Compared to the original accuracy, DDASR, DASR, and deep learning-based baselines have on average decreased the restored veritable accuracy by 3.81%, 6.09%, and 12.36% on the diverse Java dataset, as well as decreased by 6.17%, 6.69%, and 13.43% on the Python dataset, respectively. This suggests that constructing the pseudo ground truth through clustering and relocating is more effective than using *<UNK>* tags for replacement.

(4) The differences among DDASR with different LLMs and among DASR applying different LLMs are not significant on accuracy ($p>0.05$). We evaluate the statistical differences among DASR with different LLM encoder-decoder architectures and among DDASR with different LLM encoder-decoder architectures. It can be found that there are no significant differences ($p>0.05$) as shown in Fig. 8 and Fig. 11. Furthermore, the results suggest that DASR and DDASR, when utilizing CodeBERT, perform the best on the diverse Java dataset, while DASR using CodeBERT and DDASR using UniXcoder excel on the Python dataset.

(5) Compared with baselines, the diversity of DASR is still low, while DDASR typically exhibits strong coverage ($p<0.05$). It demonstrates that DDASR, which employs more tail APIs than baselines and DASR focusing on non-tail APIs, can improve the diversity of API sequence recommendation tasks. In consequence, DDASR can significantly enhance diversity ($p<0.05$) as shown in Fig. 9(b) and Fig. 12(b) while maintaining the accuracy ($p>0.05$) as shown in Fig. 10 and Fig. 13.

Answer to RQ2: DDASR exhibits superior diversity compared to DASR and the baselines. In terms of accuracy, there is no significant difference between DDASR and DASR. However, DDASR does outperform the baselines. These findings indicate that our DDASR can significantly increase diversity without compromising accuracy.

4.6.3 *RQ3. Can DDASR help programmers address tasks more effectively?*

Motivation. The ultimate goal of API sequence recommendation is to assist developers. Thus, in this section, we conduct a human evaluation to assess our DDASR.

Approach. We randomly select 100 queries and generated top-10 API sequences by DDASR on the diverse dataset to evaluate the validity of the results for programmers. Six developers who study or work in computer-related fields with experience in Java projects are invited to join a subject group. Three of the developers are currently pursuing master's degrees with more than two years of experience programming in Java, while the other three are engaged in Java development in Internet companies, such as Alibaba, Byte Jump, and Huawei. It is important to note that they are not co-authors. They independently give a rating score for the generated API sequences according to their experience. A score of zero indicates that they believed the API sequences generated are completely useless for the query, while a score of one denotes they are valuable to the developers, for example, the key API for solving the query appears in the generated API sequences or the results inspire them. The rating score is two if the API sequences generated are a way to the query. In addition, the developers are also allowed to search the Internet for unfamiliar concepts.

Table 6. The rating scores of the human evaluation. A score of zero indicates that the generated API sequences are of no help in solving the requirement, a score of one indicates it is valuable, and a score of two suggests it can solve

Developers	# Rating Score Zero	# Rating Score One	# Rating Score Two	Average Rating Score
Developer 1	15	36	49	1.34
Developer 2	22	36	42	1.20
Developer 3	19	28	53	1.34
Developer 4	5	56	39	1.34
Developer 5	2	32	66	1.64
Developer 6	4	32	64	1.60

Results. The results of the human evaluation are shown in Table 6. Developers 1-3 are students and developers 4-6 have working experience in Internet companies. Table 7 shows ten queries with the same rating scores and their ground truth, top-3 generated API sequences, and human rating scores. These queries vary in length, containing both very long and short queries that were not included in the training set. They include requirements for page layout, mathematical calculation, string processing, file streaming, and so on. In addition, in the table, tail APIs are highlighted in italics.

(1) DDASR is effective in recommending appropriate API sequences for developers in the majority of cases. The evaluation results from six developers, as presented in Table 6, suggest that DDASR is capable of recommending API sequences that either directly address the query or offer substantial assistance in most cases. Notably, developers 4-6 receive fewer zero ratings compared to developers 1-3. This discrepancy is likely attributable to the former group's higher level of programming expertise, enabling them to more effectively identify useful APIs.

(2) DDASR can generate more accurate API sequences for length queries, while its performance may degrade if the query contains infrequent words represented as *<UNK>* tags. DDASR recommends correct results for length queries like *Query 4*. *Query 5* is also lengthy, with only one API hitting, which cannot solve the query exhaustively. We assume this is because some words, which occur less frequently in the query, are replaced with *<UNK>* tags during data preprocessing. These words may be domain-specific keywords, thus preventing mining the true intent of the query when training.

(3) For short queries, DDASR tends to generate correct results, especially when the query contains words related to the API sequence or the topic of the query is precise. The generated API sequences will get high accuracy if the query contains words related to the API sequence, like the class name. *Query 6*, "*end operation*" contains the "*Operation*" class in the API sequence of the ground truth, and the generated API sequences contain all APIs in the first two of results. For queries that focus on a specific topic, like Query 3, which is dedicated to calculating the standard variance and exhibits distinct mathematical characteristics, DDASR successfully identifies the correct APIs in its second result. However, it should be noted that the sequence of API calls in this instance is not in the correct order. There are two cases whose query or correct API sequence in the ground truth contains an operation on "*String*" in Table 7. The APIs about "*String*" account for a relatively large portion of the dataset. For queries that involve only processing of *String*, DDASR can generate accurate API sequences. String is only a class used indirectly, like *Query 2*. The most critical APIs are not recommended due to the objective factor of popularity differences.

(4) Recommending cluster centers with similar functionality to tail APIs by DDASR can provide valuable assistance to developers. The ground truth of the API sequence for *Queries 8–10* includes tail APIs. For example, regarding *Query 8*, "*DTMNodeProxy.getElementsByTagNameNS*" is a tail API in the ground truth, and DDASR generates "*DTMNodeProxy.getElementsByTagName*", another tail API, for the query. Due to their similar functionality, the two tail APIs have

Table 7. Results of human evaluation with API sequences in the ground truth, top-3 recommendation results by DDASR, and human rating scores with ten queries

ID	Query	API sequence in the ground truth	API sequences generated by DDASR	Rating Score
1	create appropriate menu item	JMenu.instance JCheckBoxMenuItem.<init> JCheckBoxMenuItem.setName JCheckBoxMenuItem.addActionListener	1 JMenuItem.<init> JMenuItem.addActionListener 2 JMenuItem.<init> JMenuItem.setMnemonic JMenuItem.addActionListener 3 JMenuItem.<init> ActionListener.<init> JMenuItem.addActionListener	1
2	build json string fields values given module	StringBuilder.<init> StringBuilder.append Map<String,List<Field>>.keySet String.cast	1 String.instance String.cast Collection.instance Collection.cast 2 StringBuilder.<init> StringBuilder.append StringBuilder.deleteCharAt StringBuilder.toString 3 String.instance String.cast Object.toString Collection.instance	0
3	calculate standard deviation	Math.sqrt Math.round	1 Math.pow Math.sqrt 2 Math.pow Math.round Math.sqrt 3 Math.pow Math.sqrt	2
4	create full classpath including existing classpath additional paths jars service files	System.getenv StringBuilder.<init> File.getAbsolutePath	1 System.getenv StringBuilder.<init> StringBuilder.append 2 StringBuilder.<init> StringBuilder.append StringBuilder.toString 3 StringBuilder.<init> StringBuilder.append File.getAbsolutePath	2
5	create returns xml *<UNK>* session element, stores *<UNK>* session details, including *<UNK>* state	Document.createElement Element.setAttribute Integer.toString	1 Element.<init> Element.setAttribute Element.addContent 2 Element.<init> String.valueOf Element.setAttribute 3 Element.<init> Element.setAttribute Element.<init>	1
6	end operation	Map<Integer,LinkedList<Operation>>.get LinkedList<Operation>.isEmpty LinkedList<Operation>.getLast	1 Map<Integer,LinkedList<Operation>>.get LinkedList<Operation>.getLast LinkedList<Operation>.removeLast 2 Map<Integer,LinkedList<Operation>>.get LinkedList<Operation>.isEmpty LinkedList<Operation>.getLast 3 Map<Integer,LinkedList<Operation>>.get LinkedList<Operation>.isEmpty LinkedList<Operation>.isEmpty	2
7	open connection given url specific http method	URL.openConnection HttpURLConnection.cast HttpURLConnection.setDoOutput	1 URL.<init> URL.openConnection HttpURLConnection.cast 2 URL.openConnection HttpURLConnection.cast HttpURLConnection.setRequestProperty 3 URL.<init> URL.getHost URL.<init>	2
8	return elements contained element given fully qualified name	*DTMNodeProxy.getElementsByTagNameNS* QName.getName ArrayList.<init> NodeList.item	1 *DTMNodeProxy.getElementsByTagName* ArrayList.<init> NodeList.getLength NodeList.item 2 *DTMNodeProxy.getElementsByTagName* ArrayList.<init> Element.instance Element.cast 3 *DTMNodeProxy.getElementsByTagName* ArrayList.<init> NodeList.item List.add	1
9	locate visible view latest adapter position	LayoutManager.getChildCount *LayoutManager.getChildAt* View.getLayoutParams	1 *LayoutManager.getChildAt* Location.getX 2 *LayoutManager.getChildAt* *LayoutManager.getChildAt* 3 *LayoutManager.getChildAt* Location.getX Location.getY	1
10	set right child indent	*AbstractLayoutCache.invalidateSizes* BasicTreeUI.updateSize	1 BasicTreeUI.updateSize ArrayList.add 2 BasicTreeUI.updateSize 3 *AbstractLayoutCache.invalidateSizes*	2

been clustered into the same cluster "*DTMNodeProxy.getElementsByTagName*" during data preprocessing. Developers will get inspiration from the suggestions provided by DDASR for APIs with features similar to the ground truth for this query. In the case of *Query 9* and *Query 10*, DDASR can successfully hit the tail APIs.

Answer to RQ3: The human evaluation, conducted by six developers, reveals that the API sequences recommended by DDASR are perceived as useful. Specifically, the developers found it advantageous to have functionally similar cluster centers suggested for tail APIs within sequences that contain tail APIs.

5 THREATS TO VALIDITY

This section discusses potential threats to the validity of our research and experimental design.

Internal Validity. Threats to internal validity, primarily concerning experiment bias and errors, manifest in aspects such as dataset construction and baseline replication [13, 42]. Firstly, the popularity difference of APIs exists objectively, which is not considered in the original open-source Java dataset. To mitigate this threat, we employ the hierarchical sampling method to build a diverse dataset with the same distribution of *query-APIseq* pairs containing tail APIs in the test set as in the training set. Additionally, we conduct multiple checks to ensure that questions in the test set are not included in the training set. Another threat to internal validity, the implementation of baseline approaches, is discussed from the different categories of baselines. For DeepAPI, CodeBERT, and CodeTrans, we directly utilize their open-source code or models. Nevertheless, we modify BIKER to make it applicable to our study. Despite this, there is somewhat of a threat to implementing BIKER and DDASR. To mitigate it, we have conducted multiple code reviews by recruiting experienced programmers and made the source code available on GitHub.

External Validity. External validity refers to the extent to which the results can be generalized beyond the scope of the study. We explore the threats to external validity, which pertain to the generalizability of DDASR. The pairs of query and API sequences in the datasets we use are based on Java and Python. Although the evaluation of two programming languages has validated the generalizability of DDASR to some extent, there is still the possibility that our model may not work well with other libraries or programming languages. To further validate the model, we will collect more data from other libraries and programming languages for training in the future.

Construct Validity. Threats to construct validity relate to the suitability of our evaluation measures. We use BLEU, a widely used metric in the translation task, to evaluate the accuracy of API sequences, because generating API sequences from a query can be analogous to translating. MAP and NDCG are the classical accuracy evaluation measures in recommender systems, which are also widely used in the software engineering field. Coverage is widely used in diversity recommendation as an evaluation metric for the proportion of recommended items. In addition, to assess the impact of the pseudo ground truth on evaluating authenticity, we introduce a restored veritable accuracy metric to mitigate bias. As a result of different coding habits, developers have different ways of implementing requirements when completing development tasks. However, there is only one correct way to solve the query in the ground truth of the dataset. We utilize a user study to mitigate this threat through manual evaluation by developers with programming experience.

Conclusion Validity. Conclusion validity concerns how much the experiment setting influences the observed result. The baseline approaches split APIs into fragments for data preprocessing and experimental evaluation, while our study regards a complete API as a basic unit for the two stages. This difference presents a potential risk to conclusion validity. To address this, we conduct an ablation experiment with DASR, a variant of our approach, assessing the impact of training solely with aggregated API fragments.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose DDASR to recommend API sequences automatically for developer queries. Our study highlights the importance of incorporating diversity in API sequence recommendation, particularly with tail APIs. DDASR can recommend functionally similar tail APIs, helping to mitigate the long-tail effect. For programming tasks, recommending APIs with similar functionality can provide heuristic assistance to developers. We achieve this by clustering tail APIs based on function and substituting them with cluster centers to create a pseudo ground truth, followed by recommendations based on Seq2Seq and learning-to-rank techniques. Due to the outstanding performance of LLMs in natural language processing tasks, we also leverage widely adopted LLMs for learning query representations. The evaluation with state-of-the-art baselines on the original Java dataset confirms DDASR's accuracy. Experiments on the diverse Java dataset and the Python dataset show that DDASR can achieve the best diversity without significantly reducing accuracy.

Considering the personalized coding styles of developers, multiple API combinations may fulfill the same requirement, suggesting that the API sequence in the ground truth is not the only correct solution. In the future, we hope to satisfy personalized and multi-objective API sequence recommendations from various aspects of dataset construction, model improvement, and innovation in evaluation metrics. We also plan to integrate a broader range of refined tail APIs into the sequences to further improve DDASR's performance. Furthermore, we intend to apply DDASR across more programming languages and a wider range of requirement domains.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (Grant No. 2022YFF0902701) and the National Natural Science Foundation of China (No. 62032016).

REFERENCES

[1] Gediminas Adomavicius and YoungOk Kwon. 2011. Improving aggregate recommendation diversity using ranking-based techniques. *IEEE Transactions on Knowledge and Data Engineering* 24, 5 (2011), 896–911. https://doi.org/10.1109/TKDE.2011.15.

[2] Gediminas Adomavicius and YoungOk Kwon. 2014. Optimization-based approaches for maximizing aggregate recommendation diversity. *INFORMS Journal on Computing* 26, 2 (2014), 351–369. https://doi.org/10.1287/ijoc.2013.0570.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2021)*. 2655–2668.

[4] Chris Anderson. 2012. *The Long Tail.* 137–152. https://doi.org/10.18574/nyu/9780814763025.003.0014.

[5] Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. 2006. Neural Probabilistic Language Models. *Innovations in Machine Learning: Theory and Applications* (2006), 137–186. https://doi.org/10.1007/3-540-33486-6_6.

[6] Steven Bird. 2006. NLTK: The Natural Language Toolkit. In *Proceedings of the COLING/ACL Interactive Presentation Sessions (ACL 2006)*. 69–72. https://doi.org/10.3115/1225403.1225421.

[7] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1998)*. 335–336. https://doi.org/10.1145/290941.291025.

[8] 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 (SIGSOFT FSE 2012)*. 1–11. https://doi.org/10.1145/2393596.2393606.

[9] Laming Chen, Guoxin Zhang, and Hanning Zhou. 2018. Fast Greedy MAP Inference for Determinantal Point Process to Improve Recommendation Diversity. arXiv:1709.05135 [cs.IR]

[10] Peizhe Cheng, Shuaiqiang Wang, Jun Ma, Jiankai Sun, and Hui Xiong. 2017. Learning to recommend accurate and diverse items. In *Proceedings of the 26th International Conference on World Wide Web (WWW 2017)*. 183–192. https://doi.org/10.1145/3038912.3052585.

[11] Sunhao Dai, Ninglu Shao, Haiyuan Zhao, Weijie Yu, Zihua Si, Chen Xu, Zhongxiang Sun, Xiao Zhang, and Jun Xu. 2023. Uncovering ChatGPT's Capabilities in Recommender Systems. In *Proceedings of the 17th ACM Conference on Recommender Systems (RecSys 2023)*. 1126–1132. https://doi.org/10.1145/3604915.3610646.

[12] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. arXiv:2104.02443 [cs.SE]

[13] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In *Proceedings of the 22nd International Conference on Software Engineering Knowledge Engineering (SEKE 2010)*. 374–379.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the Findings of the Association for Computational Linguistics (EMNLP Findings 2020)*. 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[15] Luke Friedman, Sameer Ahuja, David Allen, Zhenning Tan, Hakim Sidahmed, Changbo Long, Jun Xie, Gabriel Schubiner, Ajay Patel, Harsh Lara, Brian Chu, Zexi Chen, and Manoj Tiwari. 2023. Leveraging Large Language Models in Conversational Recommender Systems. arXiv:2305.07961 [cs.IR]

[16] Junchen Fu, Fajie Yuan, Yu Song, Zheng Yuan, Mingyue Cheng, Shenghui Cheng, Jiaqi Zhang, Jie Wang, and Yunzhu Pan. 2023. Exploring Adapter-based Transfer Learning for Recommender Systems: Empirical Studies and Practical Insights. arXiv:2305.15036 [cs.IR]

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

[18] Marko Gasparic and Andrea Janes. 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software* 113 (2016), 101–113. https://doi.org/10.1016/j.jss.2015.11.036.

[19] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2016)*. 631–642. https://doi.org/10.1145/2950290.2950334.

[20] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL 2022)*. 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499.

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations (ICLR 2021)*.

[22] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2016. Session-based Recommendations with Recurrent Neural Networks. arXiv:1511.06939 [cs.LG]

[23] Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. 2023. Large Language Models are Zero-Shot Rankers for Recommender Systems. arXiv:2305.08845 [cs.IR]

[24] 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 (ASE 2018)*. 293–304. https://doi.org/10.1145/3238147.3238191.

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG]

[26] Zhengbao Jiang, Zhicheng Dou, Wayne Xin Zhao, Jian-Yun Nie, Ming Yue, and Ji-Rong Wen. 2018. Supervised Search Result Diversification via Subtopic Attention. *IEEE Transactions on Knowledge and Data Engineering* 30, 10 (2018), 1971–1984. https://doi.org/10.1109/TKDE.2018.2810873.

[27] Marius Kaminskas and Derek Bridge. 2016. Diversity, Serendipity, Novelty, and Coverage: A Survey and Empirical Analysis of Beyond-Accuracy Objectives in Recommender Systems. *ACM Transactions on Interactive Intelligent Systems* 7, 1 (2016), 1–42. https://doi.org/10.1145/2926720.

[28] Wang-Cheng Kang and Julian McAuley. 2018. Self-Attentive Sequential Recommendation. In *IEEE International Conference on Data Mining (ICDM 2018)*. 197–206. https://doi.org/10.1109/ICDM.2018.000356.

[29] Yejin Kim, Kwangseob Kim, Chanyoung Park, and Hwanjo Yu. 2019. Sequential and Diverse Recommendation with Long Tail. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. 2740–2746. https://doi.org/10.24963/ijcai.2019/380.

[30] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *IEEE Symposium on Visual Languages - Human Centric Computing (VL/HCC 2004)*. 199–206. https://doi.org/10.1109/VLHCC.2004.47.

[31] Philipp Koehn. 2004. Pharaoh: A Beam Search Decoder for Phrase-Based Statistical Machine Translation Models. In *Machine Translation: From Real Users to Research (AMTA 2004)*. 115–124. https://doi.org/10.1007/978-3-540-30194-3_13.

[32] Xianglong Kong, Weina Han, Li Liao, and Bixin Li. 2020. An analysis of correctness for API recommendation:are the unmatched results useless? *Science China Information Sciences* 63 (2020), 1–15. https://doi.org/10.1007/s11432-019-2929-9.

[33] Hyokmin Kwon, Jaeho Han, and Kyungsik Han. 2020. ART (Attractive Recommendation Tailor): How the Diversity of Product Recommendations Affects Customer Purchase Preference in Fashion Industry?. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM 2020)*. 2573–2580. https://doi.org/10.1145/3340531.3412687.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539.

[35] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[36] Jing Li, Pengjie Ren, Zhumin Chen, Zhaochun Ren, Tao Lian, and Jun Ma. 2017. Neural attentive session-based recommendation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM 2017)*. 1419–1428. https://doi.org/10.1145/3132847.3132926.

[37] Shuang Li, Yuezhi Zhou, Di Zhang, Yaoxue Zhang, and Xiang Lan. 2017. Learning to Diversify Recommendations Based on Matrix Factorization. In *IEEE 15th International Conference on Dependable, Autonomic and Secure Computing, 15th International Conference on Pervasive*

Intelligence and Computing, 3rd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech 2017). 68–74. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.26.

[38] Jianghao Lin, Xinyi Dai, Yunjia Xi, Weiwen Liu, Bo Chen, Xiangyang Li, Chenxu Zhu, Huifeng Guo, Yong Yu, Ruiming Tang, and Weinan Zhang. 2023. How Can Recommender Systems Benefit from Large Language Models: A Survey. arXiv:2306.05817 [cs.IR]

[39] Junling Liu, Chao Liu, Peilin Zhou, Renjie Lv, Kang Zhou, and Yan Zhang. 2023. Is ChatGPT a Good Recommender? A Preliminary Study. arXiv:2304.10149 [cs.IR]

[40] James Martin and Jin L. C. Guo. 2022. Deep API Learning Revisited. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC 2022)*. 321–330. https://doi.org/10.1145/3524610.3527872.

[41] 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 (ICSE 2011)*. 111–120. https://doi.org/10.1145/1985793.1985809.

[42] Nasser Mustafa, Yvan Labiche, and Dave Towey. 2019. Mitigating Threats to Validity in Empirical Software Engineering: A Traceability Case Study. In *IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC 2019)*. 324–329. https://doi.org/10.1109/COMPSAC.2019.10227.

[43] Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22 (2017), 259–291. https://doi.org/10.1007/s10664-015-9421-5.

[44] Yoon-Joo Park. 2012. The adaptive clustering method for the long tail problem of recommender systems. *IEEE Transactions on Knowledge and Data Engineering* 25, 8 (2012), 1904–1915. https://doi.org/10.1109/TKDE.2012.119.

[45] Yoon-Joo Park and Alexander Tuzhilin. 2008. The Long Tail of Recommender Systems and How to Leverage It. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys 2008)*. 11–18. https://doi.org/10.1145/1454008.1454012.

[46] Massimo Quadrana, Alexandros Karatzoglou, Balázs Hidasi, and Paolo Cremonesi. 2017. Personalizing session-based recommendations with hierarchical recurrent neural networks. In *Proceedings of the 11th ACM Conference on Recommender Systems (RecSys 2017)*. 130–137. https://doi.org/10.1145/3109859.3109896.

[47] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean - Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. 357–367. https://doi.org/10.1145/2884781.2884808.

[48] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. 2016. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. 349–359. https://doi.org/10.1109/SANER.2016.80.

[49] Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2010. Factorizing personalized markov chains for next-basket recommendation. In *Proceedings of the 19th International Conference on World Wide Web (WWW 2010)*. 811–820. https://doi.org/10.1145/1772690.17727738.

[50] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (2010), 80–86. https://doi.org/10.1109/MS.2009.161.

[51] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents. arXiv:2304.09542 [cs.CL]

[52] Jiaxi Tang and Ke Wang. 2018. Personalized Top-N Sequential Recommendation via Convolutional Sequence Embedding. In *Proceedings of the 11th ACM International Conference on Web Search and Data Mining (WSDM 2018)*. 565–573. https://doi.org/10.1145/3159652.3159656.

[53] Von Luxburg Ulrike. 2007. A Tutorial on Spectral Clustering. *Statistics and Computing* 17, 4 (2007), 395–416. https://doi.org/10.1007/s11222-007-9033-z.

[54] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2021)*. 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685.

[55] Jacek Wasilewski and Neil Hurley. 2016. Incorporating diversity in a learning to rank recommender system. In *The 29th International Flairs Conference (FLAIRS 2016)*. 572–578.

[56] Mark Wilhelm, Ajith Ramanathan, Alexander Bonomo, Sagar Jain, Ed H. Chi, and Jennifer Gillenwater. 2018. Practical Diversified Recommendations on YouTube with Determinantal Point Processes. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM 2018)*. 2165–2173. https://doi.org/10.1145/3269206.3272018.

[57] Chao-Yuan Wu, Amr Ahmed, Alex Beutel, Alexander J Smola, and How Jing. 2017. Recurrent recommender networks. In *Proceedings of the 10th ACM International Conference on Web Search and Data Mining (WSDM 2017)*. 495–503. https://doi.org/10.1145/3018661.3018689.

[58] Haolun Wu, Yansen Zhang, Chen Ma, Fuyuan Lyu, Bowei He, Bhaskar Mitra, and Xue Liu. 2023. Result Diversification in Search and Recommendation: A Survey. arXiv:2212.14464 [cs.IR]

[59] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise Approach to Learning to Rank: Theory and Algorithm. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*. 1192–1199. https://doi.org/10.1145/1390156.1390306.

[60] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. 2012. Challenging the Long Tail Recommendation. *Proceedings of the VLDB Endowment* 5, 9 (2012), 896–907. https://doi.org/10.14778/2311906.2311916.

[61] Fajie Yuan, Alexandros Karatzoglou, Ioannis Arapakis, Joemon M Jose, and Xiangnan He. 2019. A simple convolutional generative network for next item recommendation. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining (WSDM 2019)*. 582–590. https://doi.org/10.1145/3289600.3290975.

[62] Jingxuan Zhang, He Jiang, Zhilei Ren, and Xin Chen. 2018. Recommending APIs for API Related Questions in Stack Overflow. *IEEE Access* 6 (2018), 6205–6219. https://doi.org/10.1109/ACCESS.2017.2777845.

[63] Qi Zhang, Jingjie Li, Qinglin Jia, Chuyuan Wang, Jieming Zhu, Zhaowei Wang, and Xiuqiang He. 2021. UNBERT: User-News Matching BERT for News Recommendation. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*. 3356–3362. https://doi.org/10.24963/ijcai.2021/462.

[64] Shuai Zhang, Yi Tay, Lina Yao, and Aixin Sun. 2018. Next Item Recommendation with Self-Attention. arXiv:1808.06414 [cs.IR]

[65] Yuhui Zhang, HAO DING, Zeren Shui, Yifei Ma, James Zou, Anoop Deoras, and Hao Wang. 2021. Language Models as Recommender Systems: Evaluations and Limitations. In *I (Still) Can't Believe It's Not Better! NeurIPS 2021 Workshop*.

[66] Yu Zheng, Chen Gao, Liang Chen, Depeng Jin, and Yong Li. 2021. DGCN: Diversified Recommendation with Graph Convolutional Networks. In *Proceedings of the Web Conference (WWW 2021)*. 401–412. https://doi.org/10.1145/3442381.3449835.

[67] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference Object-Oriented Programming (ECOOP 2009)*. 318–343. https://doi.org/10.1007/978-3-642-03013-0_15.

[68] Jianghong Zhou, Eugene Agichtein, and Surya Kallumadi. 2020. Diversifying Multi-Aspect Search Results Using Simpson's Diversity Index. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM 2020)*. 2345–2348. https://doi.org/10.1145/3340531.3412163.

[69] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. 2005. Improving Recommendation Lists through Topic Diversification. In *Proceedings of the 14th International Conference on World Wide Web (WWW 2005)*. 22–32. https://doi.org/10.1145/1060745.1060754.