PICASO: Enhancing API Recommendations with Relevant Stack Overflow Posts

Ivana Clairine Irsan, Ting Zhang*, Ferdian Thung, Kisub Kim, and David Lo

School of Computing and Information Systems, Singapore Management University

Email: {ivanairsan, tingzhang.2019, ferdianthung, kisubkim, davidlo}@smu.edu.sg

Abstract—While having options could be liberating, too many options could lead to the sub-optimal solution being chosen. This is not an exception in the software engineering domain. Nowadays, API has become imperative in making software developers' life easier. APIs help developers implement a function faster and more efficiently. However, given the large number of open-source libraries to choose from, choosing the right APIs is not a simple task. Previous studies on API recommendation leverage natural language (query) to identify which API would be suitable for the given task. However, these studies only consider one source of input, i.e., GitHub or Stack Overflow, independently. There are no existing approaches that utilize Stack Overflow to help generate better API sequence recommendations from queries obtained from GitHub. Therefore, in this study, we aim to provide a framework that could improve the result of the API sequence recommendation by leveraging information from Stack Overflow. In this work, we propose PICASO, which leverages a bi-encoder to do contrastive learning and a cross-encoder to build a classification model in order to find a semantically similar Stack Overflow post given an annotation (i.e., code comment). Subsequently, PICASO then uses the Stack Overflow's title as a query expansion. PICASO then uses the extended queries to fine-tune a CodeBERT, resulting in an API sequence generation model. Based on our experiments, we found that incorporating the Stack Overflow information into CodeBERT would improve the performance of API sequence generation's BLEU-4 score by 10.8%.

Index Terms—API recommendation, Multi-source analytics, Sequence Generation, Pre-trained Models, Stack Overflow, Query Expansion

I. INTRODUCTION

In modern software development, Application Programming Interfaces (APIs) and libraries have been widely used by developers to improve development efficiency. Considering the large volume of existing APIs, multiple APIs can be used to implement the same functionality. For instance, to read a file in Java using the JDK library, a developer could use either FileReader, BufferedReader, Files, Scanner, etc., depending on the size of the file and type of the data that need to be read (i.e., byte or characters). While having many options is liberating, the problem of choosing the right API for a certain functionality arises. Given the enormous number of available APIs, it is challenging for developers to find and learn which API to use along with the complementary APIs needed in the API invocation sequence [1]. Therefore, it is desirable to have an API recommendation system that can recommend the correct APIs to use [2], [3].

To support developers in finding the correct APIs, several API recommendation methods have been proposed [4]–[7]. Depending on the input type, these methods can be categorized into query-based or code-based API recommendation [1]. In this work, we mainly focus on the former type of API recommendation. We are concerned with how to accurately translate a natural language query that describes the intention of the programming task to an API sequence that implements the requirements. While many query-based API recommendation approaches have been proposed [4]–[6], only a limited number of them recommend API sequence [2], [8]. The best of which is the work by Martin and Guo [8] that leverages CodeBERT, a bimodal pre-trained model for programming language and natural language, to generate API sequences.

In practice, to search for suitable APIs, developers tend to resort to Stack Overflow [4]. Prior studies [4]–[6] have shown that Stack Overflow can be effectively utilized to recommend a set of APIs as it contains tremendous crowd-knowledge. Additionally, diversifying the resources for training an API recommendation model has been investigated by researchers [6], [9]–[11] and verified to be beneficial.

Previous studies [4], [6] utilize Stack Overflow data by finding similar Stack Overflow posts and recommending curated APIs mentioned in the answers. Their objective is to correctly identify the similar Stack Overflow posts for a question that a developer comes up with. However, developers' need for good API recommendations goes beyond a question when they face difficulties. On a daily basis, they implement functions to perform certain tasks that could be summarized in the form of natural language (NL), i.e., code comment/annotation. Oftentimes, the NL representation of the code could be paired with a Stack Overflow post (example presented in Section II-B). This motivates us to explore the potential of leveraging Stack Overflow information in search of generating better API sequence recommendations.

With a goal to ease developers' daily tasks, we propose PICASO (Enhancing API Recommendations with Relevant Stack Overflow Posts), an API recommendation technique that leverages multi-source (i.e., Stack Overflow and GitHub) information via query expansion. PICASO takes a user query as either code comment or annotation, then finds the Stack Overflow post that is most similar to the query. Once it discovers a similar post, the post is delivered to the CodeBERT encoder as query expansion to generate the API sequence that implements the functionality described in the query. To find the

most similar Stack Overflow post for a given annotation, we adapted a framework proposed by Wei et al. [6], who propose a state-of-the-art technique in finding similar Stack Overflow posts for a Stack Overflow query. The framework consists of a filtering and a re-ranking model. The filtering model is built on top of the contrastive learning method, while the re-ranking model is trained by performing joint-embedding training.

We evaluated PICASO on a dataset derived from the work of Martin and Guo [8]. PICASO manages to outperform the state-of-the-art API sequence recommendation model by 10.8% in terms of BLEU-4 score.

Our contributions can be summarized as follows:

- We are the first to show that Stack Overflow posts can boost the effectiveness of the query-based API sequence recommendation, particularly a query expansion.
- We propose PICASO, a multi-source API sequence recommendation method that can outperform the state-of-the-art approach.
- We conduct experiments to demonstrate the effectiveness of PICASO, achieving a 10.8% improvement measured in BLEU-4 score.

The remainder of the paper is organized as follows. Section II introduces the problem formulation and a motivating example. We describe our approach in Section III. Section IV presents our experimental setup and the research questions. Experimental results are described in Section V. We discuss the results in Section VI. Section VII presents the related works. We conclude our work and present future work in Section VIII.

II. PRELIMINARIES

A. Problem Formulation

Following prior works [2], [8], we formulate the API recommendation task as a sequence-to-sequence task. Given a query in the form of annotation, i.e., code comment that describes the purpose of a function, our goal is to recommend a sequence of API calls that implements the functionality described in the query. We also aim to leverage information from multiple sources such as Stack Overflow and GitHub.

Here, we formally define the task of query-based API recommendation. We denote the input sequence, e.g., annotation, as $W = \langle W_1, W_2, \ldots, W_n \rangle$, where $W_i = \langle w_i^1, w_i^2, \ldots, w_i^{n_i} \rangle$ and w_i^j refers to the jth subtoken in word W_i. We want to generate an API sequence to be recommended. We denote the API sequence as $A = \langle A_1, A_2, \ldots, A_m \rangle$, where $A_i = \langle s_i^1, s_i^2, \ldots, s_i^{n_i} \rangle$. s_i^j refers to the jth subtoken in API call A_i.

When a model predicts the subtoken s_i^1, the input of the model would be the input sequence $\langle W_1, W_2, \ldots, W_n \rangle$ and the previously predicted subtokens $\langle s_1^1, \ldots, s_1^{n_1}, \ldots, s_{i-1}^1, \ldots, s_{i-1}^{n_{i-1}} \rangle$. When the end token is predicted, an API recommendation model will output the chain of predicted subtokens as the API sequence. Take the annotation and the API sequence in the first box in Figure 4 as an example. In the beginning, the input is only the annotation. After the API recommendation model predicts

two subtokens in the target API sequence, the input will be $\langle W_1, W_2, W_3, W_4, s_5^1, s_5^2 \rangle$, where $W_1 = parse$, $W_2 = string$, $W_3 = to$, $W_4 = object$, $s_5^1 = Float$, $s_5^2 = ..$. The API recommendation model would predict the next token based on this input. Suppose the model predicts $parseFloat$ and $\langle \texttt{eos} \rangle$, the output API sequence would be the combination of these three subtokens s_5^1, s_5^2 and $parseFloat$.

B. Motivating Example

We use the example in Figure 4 to motivate the need for leveraging Stack Overflow posts for API sequence recommendation. The annotation for this example is `parse string to object`, and the target APIs are `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, and `Double.parseDouble`. From the annotation alone, it is not possible to know that the input `String` contains numbers, and the `Object` should be an instance of numbers (e.g., `Integer`). Broadly speaking, the input `String` can contain any set of characters, and the output `Object` can be an instance of any object.

To obtain a more specific query, we can expand based on the Stack Overflow posts that may contain similar sets of APIs. We define the similarity between an annotation and a Stack Overflow post as the proportion of APIs from the target APIs that are mentioned in the post. The more APIs from the target APIs that are mentioned in the Stack Overflow post, the more similar the Stack Overflow post is to the annotation. Based on this definition, we can prepare the training data which contains the most similar Stack Overflow post and the annotation; and learn a similarity function from it. In practice, in which the target APIs are not available, we can use the learned similarity function to find the most similar posts given the annotation. We describe in detail how we construct such a function in Section III-A.

Consider that we have a collection of Stack Overflow posts containing SO Post 1, SO Post 2, and SO Post 3 as shown in Figure 4. Among these Stack Overflow posts, SO Post 1 would be the most similar since it contains the largest number of relevant APIs: `Float.parseFloat`, `Integer.parseInt`, and `Double.parseDouble`. The title for SO Post 1 is `parse String containing a Number into a INT`. By expanding the query with this title, we can now understand that the input `String` should be the numbers.

The above illustration demonstrates that we can leverage Stack Overflow to improve the API sequence recommendation, particularly by adding more information from the Stack Overflow so that we can input a more accurate query to the API sequence generation model.

III. APPROACH

In this section, we introduce the overall architecture of PICASO and discuss the details of each component. The overall architecture is presented in Figure 1.

In general, PICASO consist of two phases: 1) **Finding Similar SO Post**. This phase is used to find Stack Overflow post

Fig. 1. PICASO high-level architecture

that is the most semantically similar to the input annotation (i.e., code comment) and 2) **API sequence generation**. This takes annotation and the Stack Overflow information retrieved from the first phase to generate an API sequence.

For the Finding Similar SO Post phase, we adapt CLEAR's [6] framework, which has been modified to leverage the sentence embedding technique for finding relevant posts. While CLEAR leverages the sentence embeddings to map a Stack Overflow post towards other similar Stack Overflow posts, we utilize sentence embedding to map an annotation towards similar Stack Overflow posts. Therefore, we leverage the usage of contrastive learning and joint embedding training while implementing our adaptation of the triplet data generation. For the second phase, we leverage CodeBERT as the base encoder for the encoder-decoder architecture since it is state-of-the-art for API sequence generation technique [8]. Further explanation is given in Section III-B.

The detailed architecture of PICASO is presented in Figure 2. It shows the choices of the model used in every component of PICASO, as well as the data flow.

A. Finding Similar SO Post

To find the most similar Stack Overflow post, we leverage contrastive learning [12] and joint embedding training [13]. The same approach has been used in CLEAR [6], with a difference in domain of the data. The objective of CLEAR is to find the most relevant Stack Overflow posts for a given Stack Overflow post question. In other words, they aim to build a better sentence representation for two natural language texts from the same domain. On the contrary, in this work, we aim to train a sentence embedding model that could place an annotation derived from GitHub's project towards Stack Overflow post, which comes from different domains.

Although both GitHub and Stack Overflow are from the software engineering domain, the nature of the content is essentially different. For example, the nature of the natural language (NL) from Stack Overflow comes in the form of a question, oftentimes related to an error rather than a technical functionality of an API. Moreover, the APIs mentioned in the answer posts are not necessarily a sequence. On the other hand, annotation in GitHub data portrays the functionality of a task in the form of a statement rather than a question. Moreover, the APIs extracted from GitHub are presented in the form of a sequence. We believe that adapting contrastive learning and joint embedding training in our approach as an attempt to bring similar annotation and Stack Overflow post together in the sentence embedding is advantageous. Sequentially, the most similar post retrieved in this phase will be used for query expansion in the API sequence generation phase.

The architecture of the framework to find the most relevant post is presented in Figure 3. The filtering and re-ranking model produced in this step will be utilized to find the most semantically similar post given an annotation. Finding a semantically relevant post is imperative, as we utilize such a post as query expansion that will directly affect the performance of our API generation model. The details for each process are explained as below:

Triplet Data Generation: In order to train both the filtering model via contrastive learning and the re-ranking model via joint embedding, we need to prepare the training and validation data that are built from the training set of our dataset. The training process takes triplet data as input containing an annotation, a positive, and a negative Stack Overflow title. To accommodate the triplet data generation, we need to find highly similar annotations and Stack Overflow pair to be used as positive pair. Following the motivation in Section II-B, we determined the relevancy of a post and annotation based on the *overlap rate* between the APIs appearing in the API sequence from GitHub (which corresponds to the annotation) and the APIs mentioned in the Stack Overflow answer. Figure 4 shows an example of similar posts, given an annotation and its corresponding API sequence. We first convert the API sequence in the training datum into a set of APIs, which we refer to as *training API set*. Towards this direction, we calculate the overlap rate of the APIs mentioned in the answer post (i.e., the proportion of APIs in the training API set). In the example, the training datum is considered similar to the SO Post 1[1] because 3 out of 4 in the training API set are mentioned in the answer post. Therefore, SO Post 1 is considered a positive training post for contrastive learning. On the other hand, SO Post 3[2] is considered a negative post as it does not contain APIs from the API training set. For this work, we use 0.75 as the threshold for the overlap rate as we consider this threshold captures high-quality data points while ensuring we have enough data points to train the Bi-Encoder (i.e., via contrastive learning) and Cross-Encoder (i.e., via joint-embedding training).

Due to this threshold, for SO Post 2[3], even though it has one matching API, it is not counted as the positive example for the training datum since the overlap rate is only 0.25, which is less than the 0.75 threshold.

Once we discover all of the positive posts for each training datum, we randomly choose five posts with a really small overlap rate (i.e., less than 0.01) and use them as negative posts. Pairs of positive and negative posts are then formed into triplets to be used in contrastive learning. Note that some training data are naturally discarded because they do not have a positive Stack Overflow post with an overlap rate above the threshold. Following a prior study [6], we used two parameters, namely *p* and *n* to build the triplets, where *p* refers

---

[1]https://stackoverflow.com/questions/22016489/parse-string-containing-a-number-into-a-int
[2]https://stackoverflow.com/questions/8706070/get-boolean-from-a-jtextfield
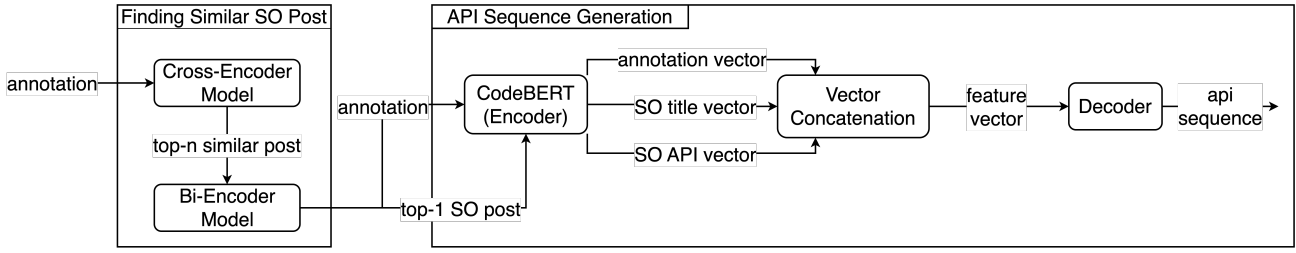[3]https://stackoverflow.com/questions/26108503/java-do-while-loops-not-looping
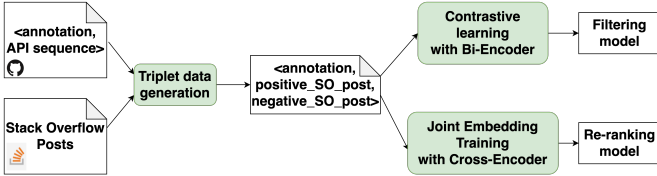
Fig. 2. PICASO detailed architecture

Fig. 3. Training process for finding the most similar Stack Overflow post
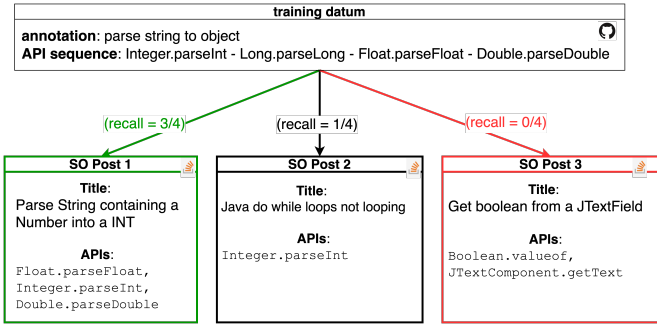

Fig. 4. Finding relevant Stack Overflow post based on API in training data and API mentioned in the SO post's answer

to the number of positive examples and *n* refers to the number of negative examples. We produced *p* x *n* numbers of triplets for each annotation *A*. Based on their grid search experiment, using *p*=10 and *n*=10 performs the best. Following this, ideally, each annotation should produce 100 triplets for contrastive learning. However, this is not the case for every annotation. If *A* only has three positive posts to be paired with, it will only have 30 triplets in the end. If *A* has more than ten positive posts, we randomly choose ten of them to generate 100 triplets for annotation *A*.

Contrastive Learning: We leverage contrastive learning [12] as the first step to learn the semantic relationship between annotation and Stack Overflow post. The contrastive training is done by fine-tuning a RoBERTa [14] based Cross-Encoder [15] model to learn a better semantic embedding representation. Even if an annotation and Stack Overflow posts do not share mutual tokens lexically, the sentence embedding should be able to put the annotation and related Stack Overflow posts in a relatively close vector space if they share the same topic semantically. The learning goal is to obtain a representative sentence embedding that could serve as a

filtering model. It should be able to effectively fetch the top-n most similar Stack Overflow posts by fetching Stack Overflow posts whose embedding has the highest cosine similarity score with the annotation embedding.

Joint Embedding Training: To train a BERT as a classification model, joint embedding training [13] is commonly used in practice. In this work, we employ a RoBERTa-based Cross-Encoder[4] model for the classification task. The purpose of the classification is to re-rank the filtered posts returned in the previous step. The training data for classification is obtained from the triplet generated for contrastive learning. We transform the triplets to make them compatible with the classification task. For each triplet, we assigned the annotation and its positive post with label `1` and we assigned label `0` for the negative post pair. The result of this joint embedding training is a classification model that can be used to perform a re-ranking for the filtered post. We leverage the probability score as a measure to determine the rank of the relevant post and take the top-1 post to be used in the API sequence generation.

Finally, after the contrastive learning and joint embedding training are done, the models produced by these learning processes will be used to pair an annotation and its target APIs with the most relevant Stack Overflow post. Using the trained cross-encoder model, we produce embeddings for the annotation and Stack Overflow posts' titles. We take the top-10 most relevant Stack Overflow posts in terms of their cosine similarity between the annotation and Stack Overflow title embedding. We then passed the top 10 posts to the bi-encoder model to be re-ranked based on the probability score. We regard the top-1 post after the re-ranking process as the most semantically relevant post for the given annotation.

B. API Sequence Generation

PICASO leverages Cross-Encoder and Bi-Encoder models trained in the previous phase to build the training data used in API sequence generation. The training data is then used to fine-tune the CodeBERT encoder-decoder model to generate an API sequence. Before the API sequence generation training phase, we leveraged the trained Cross-Encoder and Bi-Encoder to obtain the Stack Overflow post that is most similar to the annotation. Given the pair of an annotation and its most similar Stack Overflow post, we extract the title and Stack

---

[4]https://huggingface.co/cross-encoder

Overflow API from the post and build a dataset that consists of *annotation, Stack Overflow title*, *Stack Overflow API set*, and *target API sequence*. To generate the API sequence, we adopt the encoder-decoder framework.

Encoder: The encoder is used to map the input sequence into a continuous vector representation. We encode the annotation and the Stack Overflow components (i.e., Stack Overflow title and API) with a Transformer-based encoder separately. We initialize the parameters of each encoder with the pre-trained CodeBERT model. Secondly, we concatenate the individual encoded vectors (i.e., annotation vector, SO title vector, and SO API vector) into a single vector referred to as the feature vector. Last, the feature vector is passed through a Transformer decoder with 6 layers, where the parameters are randomly initialized.

Decoder: The decoder is used to generate the API sequence. The decoding starts with a start token. At each position, the decoder predicts the target token by taking the encoder output and previously generated output token list as input. The decoder stops decoding when it generates the end token.

Beam Search. During the decoding stage, we aim to generate an accurate API sequence. However, it is hard to guarantee the correctness of each subtoken at each prediction position. If the model makes a wrong prediction at a certain position, it will affect the following predictions. Thus, instead of considering only a single subtoken every time, one can consider all the subtokens in each position. The obvious drawback is the explosion of all the combinations. To achieve the balance between effectiveness and efficiency, we adopt the widely-used beam search [16], [17] to search for the recommended API sequence heuristically. Simply put, in each prediction, beam search considers the top-k subtokens (k refers to the size of the beam) based on conditional probability. Following the existing works [16], [18], we adopt the Equation 1 to calculate the chain probability of a chain of the subtokens:

$$P\left(s_m^1, \ldots, s_m^i \mid w_1^1, \ldots, w_1^{n_1}, \ldots, w_{m-1}^{n_{m-1}}\right) = \prod_{j=1}^{i} p\left(s_m^j\right)$$

(1)

where $p(s_m^j)$ refers to $p(s_m^j \mid w_1^1, \ldots, w_1^{n_1}, \ldots, w_{m-1}^{n_{m-1}}, s_m^1, \ldots, s_m^{j-1}))$. It represents the probability of the jth subtoken in the chain of $(s_m^1, \ldots, s_m^i)$.

IV. EXPERIMENTAL SETTING

In this section, we elaborate on our experiment setup, including the dataset, baseline approach, and evaluation metrics. Next, we present our experimental results that answer a few research questions.

A. Dataset

We use the dataset provided by Martin and Guo [8], which is made of pairs of annotation and API sequences. It is derived from the DeepAPI dataset [2] and was cleaned by removing duplicates. The DeepAPI dataset contains more than 7 million pairs of annotation and API sequences for training and 10,000

pairs for testing. However, Martin and Guo found that there are duplicates in the dataset. There were also some pairs in the test that appear in the training. After removing the duplicates, there are 1,880,472 training pairs and 2,441 test pairs remained. As for the Stack Overflow dataset, we utilized the dataset used by CLEAR [6]. This dataset originated from BIKER [4], containing Stack Overflow posts that are related to Java JDK programming topic, have a positive score, and have at least one accepted answer with API entities mentioned in it.

In this paper, we built our dataset by using the deduplicated DeepAPI dataset as the starting point. We combined both 1,880,472 training pairs and 2,441 test pairs and further processed them as follows:

- **Stack Overflow API selection**
  In this phase, we gathered APIs mentioned in the Stack Overflow posts answer and counted their frequency. Following that, we filtered out Stack Overflow API that is mentioned less than five times in the entire Stack Overflow posts. This ensures that the APIs appear at least 5 times, which should provide enough samples for the model to learn properly. In total, we obtained 1,398 API methods that constitute the *API vocabulary* in our dataset.

- **Dataset filtering**
  Based on the API vocabulary constructed in the previous step, we filtered out pairs of annotation and API sequences. We removed pairs whose target API contains API(s) that are not within the API vocabulary. This action ensures that the pairs share the same vocabulary with the Stack Overflow dataset. Thus, we can evaluate whether considering multiple sources of information helps (or not) when both sources of information contain information about specific APIs.

In total, we collected 196,276 pairs of annotation and API sequences in our dataset. We then split the dataset into the train, validation, and test sets with a ratio of 8:1:1. All in all, we have 157,020 pairs in the training set, 19.628 pairs in the validation set, and 19,628 pairs in the test set.

B. Baseline

Based on the previous work conducted by Martin and Guo [8], CodeBERT is the state-of-the-art approach for generating API sequences. It surpassed DeepAPI [2], the former state-of-the-art in the domain of API sequence generation. Therefore, we chose to adapt CodeBERT as our baseline model. All experiments were done with the same CodeBERT structure, which is adapted from CodeBERT project [19]. CodeBERT is a pre-trained model built specifically for solving tasks related to programming languages. It is trained on 6 different natural language and programming language pairs, resulting in a powerful pre-trained model whose embedding is proven useful for varying downstream tasks [20], [21].

C. Hyperparameter Setting

We used the same hyperparameters settings in all of our experiments. We set the maximum token length of the input (i.e., annotation) and the target sequence to be 64. For the

CodeBERT model that involves multiple inputs (e.g., Stack Overflow title and API), we also set the maximum token length of 64 for each input. All models were trained for 30 epochs.

D. Evaluation

In order to measure the performance of API sequence generation, we employ BLEU score [22] as the selected metric. We compare the BLEU score of PICASO and the baseline to identify which one is the better approach in generating API sequence recommendations. BLEU-4 score is capable of gauging how accurate a sequence that is generated by a model is compared to the correct target sequence (i.e., ground truth). As a metric that is widely adopted in machine translation problems, BLEU score is relevant to be used in comparing the automatically generated API sequence against the human-written API sequence and has been used in existing work in API sequence recommendation [2], [8], [23].

BLEU score is expressed mathematically as below:

$$BLEU = BP \times exp\left(\sum_{n=1}^{N} w_n log\left(p_n\right)\right) \quad (2)$$

$$BP = \begin{cases} 1 & c \geqslant r \\ exp\left(1 - \frac{r}{c}\right) & c < r \end{cases} \quad (3)$$

In Equation 2, BP refers to the brevity penalty. This variable aims to give a penalty to the generated sequence that is shorter than the ground truth. r and c are correlated to the number of tokens in ground truth and candidate, respectively. Furthermore, p_n is the the modified precision for n-gram, w_n are the weight and $\sum_{n=1}^{N} w_n = 1$.

For the BLEU score, the higher it is, the better the result. In this work, we measure the performance of PICASO on the test data with cumulative n-gram BLEU score with n=4. The cumulative score in the BLEU scores formula correlates to the individual n-gram scores at all orders from 1 to n. It then used the pre-defined weight to calculate the weighted geometric mean to obtain the final BLEU-n score.

E. Research Questions

In this work, we would like to investigate the following Research Questions (RQs):

- **RQ1:** *Can we improve the performance of API sequence recommendation by leveraging information from Stack Overflow for query expansion?*
  While incorporating more information tends to generate a better result, it may not always be the case. In order to answer this RQ, we compare our baseline model with a model that is built with additional information from Stack Overflow. We take CodeBERT with annotation input as the baseline and then experimented with Stack Overflow title and Stack Overflow API usage. The hypothesis for this question is that if we could find a semantically similar Stack Overflow post for an annotation, information stored in the Stack Overflow post would be imperative for the API sequence generation.

Approach	BLEU-1	BLEU-2	BLEU-3	BLEU-4
CodeBERT	0.49015	0.39946	0.33005	0.26296
PICASO	**0.51371**	**0.42767**	**0.35959**	**0.29131**

In this work, we performed training on 2 types of data. One is where we only use annotation as the query to predict API sequence (i.e., the CodeBERT baseline), and the other one is where we incorporate Stack Overflow posts information such as title and APIs as query expansion (i.e., PICASO). Both models are then evaluated by comparing the BLEU score of the generated API sequence on the test set.

- **RQ2:** *How should we utilize the information stored in Stack Overflow post to improve API sequence generation performance?*
  To answer this research question, we dig into the two types of Stack Overflow information available in our Stack Overflow dataset. Two components are being observed, namely the Stack Overflow title and Stack Overflow API. Note that we deliberately omitted Stack Overflow body from this work due to the lengthy nature of the Stack Overflow body. For the Stack Overflow API, we utilize the extracted APIs mentioned in the accepted answer and treat them as related API tokens for a Stack Overflow natural language (i.e., Stack Overflow title). We explore the impact of these two components on the API sequence generation's performance. Our hypothesis for this question is that using more information from Stack Overflow will accommodate a better API sequence generation.

V. RESULT

In this section, we show the experimental results and answer the RQs.

A. Answer to RQ1:

As shown in Table I, in general, PICASO managed to outperform the vanilla CodeBERT by 10.8% in terms of BLEU-4 score. This shows that utilizing Stack Overflow posts as query expansion that introduces additional information give better results than using only annotation in generating API sequence recommendation. The improvement is also consistent across the different BLEU scores.

We also performed Mann-Whitney U hypothesis testing and got p-value less than 0.01, indicating that PICASO is statistically significantly better than the baseline.

> **Answer to RQ1:** PICASO achieves the highest BLEU-4 score of 0.29131, which outperforms the baselines by CodeBERT by 10.8%.

Input	BLEU-1	BLEU-2	BLEU-3	BLEU-4
annotation (baseline)	0.49015	0.39946	0.33005	0.26296
annotation + SO title	0.50949	0.42230	0.35300	0.28507
annotation + SO title + SO API	**0.51371**	**0.42767**	**0.35959**	**0.29131**

B. Answer to RQ2:

We performed an ablation study to measure the impact of leveraging Stack Overflow title and Stack Overflow APIs for query expansion. Table II displays the BLEU scores for each model that is built with different Stack Overflow input(s) as the query. It shows that leveraging Stack Overflow information as query expansion can achieve a BLEU-4 score of up to 0.29131. The highest BLEU-4 score is achieved by utilizing both Stack Overflow title and Stack Overflow API in the query. Nonetheless, adding only Stack Overflow title already improved the BLEU-4 score from 0.263 to 0.285, which translates to over 8% improvement. This shows that Stack Overflow information brings a noticeable improvement when it is incorporated as query expansion.

> **Answer to RQ2:** Using Stack Overflow title and Stack Overflow API brings more improvement towards the API sequence generation technique compared to using only the Stack Overflow title.

VI. DISCUSSION

A. Ability to Find Similar Stack Overflow Post

We counted the number of cases in which the framework can successfully link an annotation to a relevant Stack Overflow post. We consider a Stack Overflow post as relevant if the target API sequence shared common API(s) with API(s) mentioned in the Stack Overflow answer post. We observed 3 degrees of success in finding a similar post, namely *All match*, *Partial match*, and *No match*. For the *All match* category, all of the APIs that are present in the target APIs also exist in the obtained in the answer post. For the *Partial match* category, a partial number of APIs are mentioned in the retrieved posts. The retrieved Stack Overflow post contains one of the target APIs in its set of APIs. We categorized this kind of match as a partial match. As for the *No match* category, none of the target APIs exist in the obtained answer post. For example, we input a pair of annotation *remove sub array* with target APIs of `Array.newInstance,System.arraycopy` and obtained a Stack Overflow post with a title of *Error making a sub-array*, which contains these APIs in the accepted answers: `System.arraycopy, Arrays.copyOfRange`. The retrieved Stack Overflow post contains `System.arrayCopy` method, which also appears in the target API.
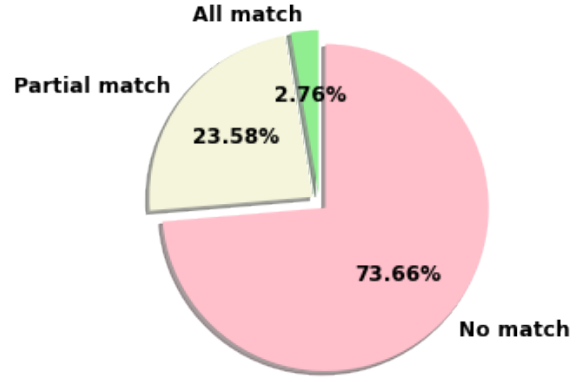

Fig. 5. Capability of finding similar Stack Overflow post for the annotations in the training data

Input	Precision	Recall
CodeBERT (baseline)	0.429	0.399
PICASO	0.452	0.442

Based on the overall result presented in Figure 5, using the Stack Overflow API in the post-processing phase, i.e., beam search, in order to limit the search space is not favorable in our case. This statement is backed by our investigation, which shows that more than 73% of the annotation in our dataset failed to find a relevant Stack Overflow post that shares common API(s) with the target API. Therefore, we deemed leveraging Stack Overflow information in the query expansion manner is more suitable for our study because even though the API method(s) are not intersected, they may still come from the same class or domain. For example, for annotation *compress array* with a target API of `ArrayList.size, Arraylist.ensureCapacity`, the retrieved Stack Overflow post is a post with *Expand array once it reaches limit* as its title, and the APIs mentioned in that post are `Arrays.copyOf` and `System.arraycopy`.

Despite only about 26% of the annotations being associated with a highly relevant Stack Overflow post, PICASO was able to increase the performance of API sequence recommendation by nearly 11%. This indicates that if similar Stack Overflow posts were linked more effectively, the performance could be further enhanced.

We also calculated precision and recall as additional metrics to supplement our analysis. While precision and recall are commonly used for evaluating API recommendation techniques, they are more appropriate for assessing API set recommendations rather than API sequence recommendations. Therefore, we primarily rely on the BLEU-4 score as the main metric in this study. The purpose of calculating the precision and recall is to determine if PICASO can provide superior

API recommendations or if it is only better at arranging the sequence of API calls compared to the baseline approach. As presented in Table III, PICASO achieves higher precision and recall scores than the baseline. The 10% improvement in the precision indicates that PICASO manages to outperform the baseline in both accurately identifying the appropriate APIs and suggesting their correct invocation order.

B. Recommendation Analysis

In order to provide the future direction of API sequence generation, we discuss the cases where PICASO fails to provide an adequate sequence as well as the successful cases. Based on our investigation, PICASO successfully incorporates Stack Overflow information to improve the API sequence generation in 7,938 cases. We also present several examples below to demonstrate how incorporating Stack Overflow information guides the sequence-to-sequence model for predicting better API sequence. Example 1 presents a case where PICASO could correctly identify the intention of the developer based on the annotation and managed to pair it with a Stack Overflow post that improves the API sequence generation performance.

In Example 1, displayed in Figure 6, the annotation indicates that the developer wants to check whether a filename matches to a filter. To do this, it could be done by using regex. Without Stack Overflow information, the baseline model predicts a completely irrelevant suggestion API sequence. It could not be used to implement the intended functionality. Nonetheless, introducing Stack Overflow information to this datum successfully points the API sequence generation model towards the correct domain. PICASO generated an API sequence that is relevant to the regex usage and is identical to the target API.

Similarly, in Example 2, the baseline produces a near identical API sequence to the target API. However, when Stack Overflow information is introduced, PICASO manages to produced a sequence that is identical to what the developer has written. It shows the capability of PICASO in generating API sequence with an assistance from Stack Overflow information.

```
Annotation:
check if the filename matches the filter

Target_API:
Pattern.compile Pattern.matcher Matcher.matches

SO_title:
How to match letters only using java regex, matches method?

SO_APIs:
Pattern.compile, String.matches

Baseline_prediction:
String.lastIndexOf, String.length

PICASO_prediction:
Pattern.compile, Pattern.matcher, Matcher.matches
```

Fig. 6. Example 1

In Example 1 and 2, PICASO manages to produce a better API sequence recommendation than the baseline. This is the advantage that is brought forward by the similar Stack Overflow post. The retrieved Stack Overflow post contains

```
Annotation:
get the prefix of the object id

Target_API:
String.indexOf String.substring

SO_title:
Java regexp substring extraction pattern

SO_APIs:
String.indexOf, String.substring

Baseline_prediction:
String.lastIndexOf, String.substring

PICASO_prediction:
String.indexOf, String.substring
```

Fig. 7. Example 2

```
Annotation:
stop server

Target_API:
Thread.isAlive, Thread.interrupt, Thread.sleep,
Socket.isclosed, Socket.close, ServerSocket.close

SO_title:
How exactly do I interrupt a thread?

SO_APIs:
Thread.currentThread, Thread.interrupted

Baseline_prediction:
ServerSocket.close, Thread.interrupt, ServerSocket.close

PICASO_prediction:
Thread.interrupt, Thread.sleep, Thread.currentThread,
Socket.isClosed, Socket.close, ServerSocket.close
```

Fig. 8. Example 3

title that is semantically relevant to the task described in the annotation, as well as informative Stack Overflow API.

Another evidence is served by Example 3, presented in Figure 8. Even though the annotation only consists of 2 words that convey an intention to stop a server, the SO title and the SO API provide an information on steps that one typically can do to stop a server. They point out that developers may need to interrupt a thread. Leveraging this information, PICASO manages to perform well, generating an output that nearly identical to the target API. On the other hand, we could see that even though the baseline contains 2 APIs from the target APIs, they are not in the correct order. It suggests the invocation of `ServerSocket.close` before invoking `Thread.interrupt`. When Stack Overflow information is introduced, the relative order between the APIs are correct. Specifically, PICASO manages to suggest the correct order of thread interruption by calling `Thread.interrupt, Thread.sleep, Socket.isclosed, Socket.close, ServerSocket.close`.

In Figure 9, we present Example 4 where PICASO produces a superior API sequence despite the relevant Stack Overflow post not having any common APIs with the baseline. In this example, the developer intends to update the state of the nodes when they add a new node. As evidenced in the target sequence, the developer implements this function with

```
Annotation:
update all the necessary state to add node to slice

Target_API:
HashMap<Long>.put, HashMap<Slice,Long>.get,
HashMap<Slice,Long>.put, HashMap<Slice>.put


SO_title:
Updating mapped values for a
TreeMap<String, TreeSet<String>>

SO_APIs:
Map.get, Map.put

Baseline_prediction:
ArrayList<Long>.add, ArrayList.add, ArrayList<Long>.add


PICASO_prediction:
HashMap<Long>.containsKey, HashMap<Long>.put,
HashMap<Slice,Long>.put, HashMap<Slice>.put
```

Fig. 9. Example 4

a HashMap library. Interestingly, even though StackOverflow API does not share a common API, the title of the post (i.e., TreeMap) directs PICASO towards the usage of Map. PICASO manages to generate a better API sequence recommendation for this case by suggesting the usage of HashMap's API, instead of ArrayList, which is suggested by the baseline. Therefore, we believe the improvement of the overall score comes from the cases where Stack Overflow post guide PICASO towards better API sequence generation.

```
Annotation:
parse the line which holds the initial heap size

Target_API:
String.indexOf, String.substring, Integer.parseInt

SO_title:
Parsing string using the Scanner class

SO_APIs:
String.split, Integer.parseInt

Baseline_prediction:
String.indexOf, String.substring, Integer.parseInt

PICASO_prediction:
String.indexOf, Integer.parseInt
```

Fig. 10. Example 5

In Example 5, PICASO performs relatively worse than the baseline. This is due to the missing API method invocation in the middle of the predicted sequence. In the target API, `String.substring` is invoked before `Integer.parseInt`. In this case, the baseline predicts the sequence correctly. However, when we introduced more information from Stack Overflow, PICASO removes an API from the sequence, resulting in a drop of BLEU-4 score. This case suggests the importance of finding a highly relevant Stack Overflow post.

All in all, in Examples 1-4, the retrieved Stack Overflow contains SO title that are highly correlated to the annotation. This leads to the success of PICASO in producing a sequence that is closer to the target API. However, in Example 5, the retrieved Stack Overflow post is less relevant as the SO title does not provide much additional information that can be used for query expansion. Rather, it may emphasizes some information over the other, which leads PICASO to generate a sequence that is further from the target API.

C. Lessons Learned

Future studies should explore methods to better bridge the semantic gap between annotation and Stack Overflow query. Even though sentence embedding can bridge the knowledge gap between Github and Stack Overflow, i.e., code annotation and Stack Overflow post, there is still a large room for improvement as 73.66% cases are *No match*. Thus, future research should explore better alternatives for doing it. If we could retrieve a truly relevant Stack Overflow post for a given annotation with a high success rate, we can also utilize the Stack Overflow information beyond query expansion methods. One option is to do a post-processing step in the generation process similar to Chen et al. work [24]. They proposed Cook, an API sequence recommendation technique that has enhanced with code-specific heuristic rules in the beam search module. As they are working with partial source code as an input, they derived several code-specific heuristic rules, for instance, syntax-oriented heuristics. Since we are working with annotation as an input and do not work with source code, we may not be able to derive a heuristic rule related to the syntax. However, it is possible to derive a rule about the sequence. For example, `Statement.close` method should be invoked after `Statement.executeUpdate` method. If we can successfully retrieve relevant Stack Overflow API for a given annotation, we could use it to limit the number of plausible APIs and use a heuristic rule to generate API sequence from them.

There exists more than one sequence to accommodate the developer's intention. We need to find a way to objectively evaluate the API sequences generated by the Seq2Seq model from the perspective of usability, i.e., effectiveness and efficiency, i.e., length of API calls. As discussed in Section VI-B, there exists more than one correct sequence to implement a functionality, as shown in Example 1. There are also several functionalities that are known to have several implementations. For example, official Java documentation page [25] published that there are at least 4 different ways to read a file in Java, each utilizing different API classes. However, there are no suitable evaluation techniques to measure the effectiveness automatically other than manual evaluation. One possible direction is to generate a dataset that contains test cases, so it could be used to judge whether the generated sequence could output a desirable result. It is a simple case though, since we cannot run the recommended API sequence directly to compare towards the test case. A code generation method needs to be applied to craft a runnable code containing, transforming a sequence of API into a runnable snippet of code.

D. Threats to Validity

Threats to internal validity are associated with aspects of our experiment. It includes the replication of the baselines and the evaluation strategy. The risk of mentioned threats was minimized by utilizing curated code and dataset from previous works. For the dataset, we derived the dataset used in our experiment from the deduplicated dataset that is provided by Martin and Guo [8]. Furthermore, we also derived the sentence embedding training from CLEAR's replication package[5]. As for the baselines, we used the same code base for CodeBERT[6]. Therefore, the risk of the contaminated dataset, incorrect re-implementation, and unfair evaluation should have been mitigated.

We consider the generalizability of our result as a threat to external validity. In this study, due to the nature of the dataset from Martin and Guo [8], we only experiment with Java programming language and are limited to API that is present in the JDK library. Moreover, we limited the API vocabulary to the APIs that are mentioned more than 5 times in the entire Stack Overflow posts.

However, we believe that the impacts should be minimal as previous work on API recommendation [2], [6], [4] also applied similar settings.

VII. Related Work

In this section, we review three lines of research that are most relevant to our work, i.e., (1) API recommendation, (2) mining API usage patterns, and (3) Mining Stack Overflow.

A. API Recommendation

In recent years, a multitude of API recommendation methods have been proposed [2], [5], [7], [11]. As mentioned in Section I, there are generally two types of methods for API recommendation. We review several recent works in this part.

***Query-based* API recommendation:** Zhou et al. [11] have proposed a framework BRAID (Boosting RecommendAtion with Implicit FeeDback) to boost the performance of query-based API recommendation systems. BRAID adopts the user selection history as feedback information. Moreover, it leverages learning-to-rank to re-rank the recommendation results. Active learning techniques have been incorporated to alleviate the "cold start" of the limited feedback information at the beginning. In addition, active learning can also speed up feedback learning. The experimental results demonstrate that this framework can boost the performance of the three API recommendation systems.

***Code-based* API recommendation:** , they solve the task as the next-token prediction task. Xie et al. [26] propose HiRec, which is based on hierarchical context. To generate hierarchical context, they use WALA [27] to generate call graphs of API methods and collect basic context from the call graphs. They, we obtain a hierarchical structure of the surrounding project-specific code. The final step is to combine both the basic context and hierarchical structure of surrounding project-specific code. Another recent approach is APIRecX [18], which was proposed to handle the out-of-vocabulary issue of cross-library API recommendation. It utilizes the GPT-based pre-trained subtoken language model. At a high level, they use byte pair encoding [28] to split each API call in each API sequence and pre-train a GPT-based language model. By fine-tuning the pre-trained model, APIRecX can recommend APIs. Different from them, our focus is to demonstrate that Stack Overflow can be leveraged to boost query-based API recommendation.

B. Mining API Usage Patterns

Other than API recommendation, a similar long-researched topic is mining API usage patterns [3], [29]–[31]. MAPO is the first algorithm that has been proposed to mine API usage patterns from source code. It was initially proposed by Xie and Pei [29] and further extended by Zhong et al. [30]. For a given query that describes the API, MAPO can leverage existing source code search engines to gather relevant source files and conduct data mining. With the mined API usage patterns, MAPO has been extended to guide programmers in locating useful code snippets [30]. Wang et al. [31] found that (1) there was a lack of metrics to measure the quality of mined API patterns, and (2) the API patterns mined by the prior approaches tend to be redundant. Therefore, they focused on addressing these issues and proposed two quality metrics, i.e., succinctness and coverage, to measure the quality of mined API patterns. Furthermore, they proposed an approach named Usage Pattern Miner (UP-Miner), which includes a two-step clustering strategy to mine succinct and high-coverage usage patterns of API methods from source code. Similarly, Fowkes et al. [3] also identified the limitation of the prior approaches: the returned API calls tend to be large, redundant, and hard to understand. To mitigate the issue, they propose PAM (Probabilistic API Miner), a near parameter-free probabilistic algorithm for mining the most interesting API call patterns.

C. Mining Stack Overflow

As Stack Overflow has been an essential part of software development, there is also rising research interest in mining Stack Overflow posts. Data from Stack Overflow posts have been extensively used for multiple purposes. Several studies have investigated leveraging Stack Overflow as complementary of additional sources for specific tasks, such as augmenting API documentation [32], maintaining code [33], and error fixing [34]. Tang et al. [33] was the first to show that comment-edit pairs in Stack Overflow can be potentially used for code maintenance. They implemented an automated approach to link comments to code-snippet edits in Stack Overflow. Furthermore, they conducted a manual investigation of statistically representative random samples of the extracted comment-edit pairs. They found that 50% of the confirmed comment-edit pairs were general and related to correction, obsolete, flaw, and extension, which can be useful for general code maintenance tasks. To demonstrate the potential use, they also leverage

---

the confirmed comment-edit pairs to submit 15 pull requests to different GitHub repositories. Among them, ten have been accepted.

Wong et al. [34] have focused on the syntax errors, and they extracted a Python dataset that contains human-made errors and their fixes from Stack Overflow. They first parse detected Python source snippet histories; second, they extract pairs of failed and fixed revisions; third, they validate pairs with interpreters; and finally, they record successfully evaluated pairs. The resulting dataset is composed of real syntax errors made by developers, so it can potentially be used for the training and evaluation of code-related tasks, such as error detection. By manual investigation, they found that errors made by Stack Overflow users do not match errors made by student developers or random mutations.

Recently, Wu et al. [35] focus on the task of identifying the relevant fragments of APIs. They propose a new approach, SO2RT, to discover relevant tutorial fragments of APIs based on Stack Overflow posts. They utilize both the labeled information (relevance between Stack Overflow Q&A pairs and APIs) and unlabeled information (tutorial fragments and APIs). They first automatically build two types of pairs, i.e., (1) relevant and irrelevant API and Stack Overflow Q&A pairs, and (2) API and tutorial fragment pairs. They then train a semi-supervised transfer learning-based relevant fragment detection model, which aims to transfer the API usage knowledge in Stack Overflow Q&A pairs to tutorial fragments. Finally, the trained model can be used to infer relevant tutorial fragments of APIs.

VIII. Conclusion and Future Work

In this work, we propose an API sequence recommendation technique, PICASO which takes a natural language description of a programming task as input. It leverages a bi-encoder to conduct contrastive learning and a cross-encoder to build a classification model for finding the most semantically similar Stack Overflow given an annotation (i.e., code comment). PICASO internally utilizes the title and APIs from similar Stack Overflow post as a query expansion to fine-tune a pre-trained model resulting in a better API sequence generation model. Based on our experimental results, we show that the title and APIs are beneficial to boost the effectiveness of API sequence recommendation. Specifically, PICASO improved approximately 11% of the BLEU-4 score for the sequence recommendation.

In the future, we plan to investigate whether Stack Overflow posts can help boost the performance of the API recommendation in other programming languages, such as Python. We also plan to utilize the information from Stack Overflow in other ways beyond query expansion. Furthermore, it is also interesting to explore the usage of other parts of Stack Overflow post such as its body to boost performance further. It may be beneficial as Stack Overflow body contains detailed information about the post.

Moreover, we plan to investigate how to use Stack Overflow posts to help with tasks beyond API recommendation, particularly those that involve natural language queries, such as bug report related tasks [36]. We also plan to investigate the challenges that developers face when using API recommendation tools, following prior works for other automated solutions, e.g., [37], [38].

Availability. Our replication package is publicly available at https://github.com/soarsmu/Picaso.

References

[1] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. Lyu, "Revisiting, benchmarking and exploring api recommendation: How far are we?" *IEEE Transactions on Software Engineering*, 2022.

[2] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.

[3] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 254–265.

[4] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[5] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[6] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: contrastive learning for api recommendation," in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, p. 376–387. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510159

[7] C. Chen, X. Peng, Z. Xing, J. Sun, X. Wang, Y. Zhao, and W. Zhao, "Holistic combination of structural and textual code information for context based api recommendation," *IEEE Transactions on Software Engineering*, 2021.

[8] J. Martin and J. L. C. Guo, "Deep api learning revisited," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 321–330. [Online]. Available: https://doi.org/10.1145/3524610.3527872

[9] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec++: An enhanced tag recommendation system for software information sites," *Empirical Software Engineering*, vol. 23, no. 2, pp. 800–832, 2018.

[10] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Prompter: A self-confident recommender system," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 577–580.

[11] Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma, and H. Gall, "Boosting api recommendation with implicit feedback," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2157–2172, 2021.

[12] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[15] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[16] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation*. Vancouver: Association for Computational Linguistics, Aug. 2017, pp. 56–60. [Online]. Available: https://aclanthology.org/W17-3207

[17] R. Shu and H. Nakayama, "Improving beam search by removing monotonic constraint for neural machine translation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 339–344. [Online]. Available: https://aclanthology.org/P18-2054

[18] Y. Kang, Z. Wang, H. Zhang, J. Chen, and H. You, "Apirecx: Cross-library api recommendation via pre-trained language model," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3425–3436.

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[20] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[21] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.

[22] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[23] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[24] C. Chen, X. Peng, B. Chen, J. Sun, Z. Xing, X. Wang, and W. Zhao, ""more than deep learning": post-processing for api sequence recommendation," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–32, 2022.

[25] "Reading, writing, and creating files." [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/io/file.html

[26] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "Hirec: Api recommendation using hierarchical context," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 369–379.

[27] "Main page - walawiki," https://wala.sourceforge.net/wiki/index.php/Main_Page, (Accessed on 12/19/2022).

[28] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1715–1725.

[29] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 54–57.

[30] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[31] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[32] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 392–403.

[33] H. Tang and S. Nadi, "On using stack overflow comment-edit pairs to recommend code maintenance changes," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–35, 2021.

[34] A. W. Wong, A. Salimi, S. Chowdhury, and A. Hindle, "Syntax and stack overflow: A methodology for extracting a corpus of syntax errors and fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 318–322.

[35] D. Wu, X.-Y. Jing, H. Zhang, Y. Zhou, and B. Xu, "Leveraging stack overflow to detect relevant tutorial fragments of apis," *Empirical Software Engineering*, vol. 28, no. 1, pp. 1–37, 2023.

[36] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 53–63.

[37] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[38] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2018.