5-2022

ARSeek: identifying API resource using code and discussion on stack overflow

Gia Kien LUONG
Singapore Management University, kiengialuong@smu.edu.sg

Mohammad HADI

Thung Ferdian
Singapore Management University, ferdianthung@smu.edu.sg

Fatemeh H. FARD

David LO
Singapore Management University, davidlo@smu.edu.sg

ARSeek: Identifying API Resource using Code and Discussion on Stack Overflow

Kien Luong[1], Mohammad Hadi[2], Ferdian Thung[1], Fatemeh Fard[2], David Lo[1]
[1]Singapore Management University, [2] University of British Columbia
{kiengialuong,ferdianthung,davidlo}@smu.edu.sg,{mohammad.hadi,fatemeh.fard}@ubc.ca

ABSTRACT

It is not a trivial problem to collect API-relevant examples, usages, and mentions on venues such as Stack Overflow. It requires efforts to correctly recognize whether the discussion refers to the API method that developers/tools are searching for. The content of the Stack Overflow thread, which consists of both text paragraphs describing the involvement of the API method in the discussion and the code snippets containing the API invocation, may refer to the given API method. Leveraging this observation, we develop ARSeek, a context-specific algorithm to capture the semantic and syntactic information of the paragraphs and code snippets in a discussion. ARSeek combines a syntactic word-based score with a score from a predictive model fine-tuned from CodeBERT. In terms of F_1-*score*, ARSeek achieves an average score of 0.8709 and beats the state-of-the-art approach by 14%.

KEYWORDS

API Resource, API Embedding, Content Classification

ACM Reference Format:
Kien Luong[1], Mohammad Hadi[2], Ferdian Thung[1], Fatemeh Fard[2], David Lo[1]. 2022. ARSeek: Identifying API Resource using Code and Discussion on Stack Overflow . In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527918

1 INTRODUCTION

Developers typically use existing libraries or frameworks to implement common functionalities. Understanding which APIs to use, the methods they offer, their distinctive names, and how to use them is vital in this regard. There may be hundreds or even thousands of APIs in a large-scale software library such as the .NET framework and JDK. Microsoft conducted a survey in 2009 in which 67.6% of respondents said that inadequate or absent resources hindered learning APIs [40].

In order to gain a deeper understanding of APIs and their usage information, developers need to inspect many web pages manually and they use automated code search tools [25, 35, 49, 53]. Stack Overflow is a commonplace for the developers to discover APIs, their simple method names, and their usages through crowd-sourced questions and answers. As many API names share simple names but provide different functionality, it is difficult to find code snippets and APIs that correspond to the specific problems posted by the developers on these platforms. Moreover, API mentions in the informal text content of Stack Overflow are often ambiguous, which makes it difficult to track down APIs and learn their uses. Therefore, we require API disambiguation to support several downstream tasks such as API recommendation [19, 39] and API mining [19, 39]. To properly index and link APIs to their related information in various sources (e.g., Stack Overflow, Javadoc, etc.), it is important to link ambiguous API mentions to their actual APIs correctly.

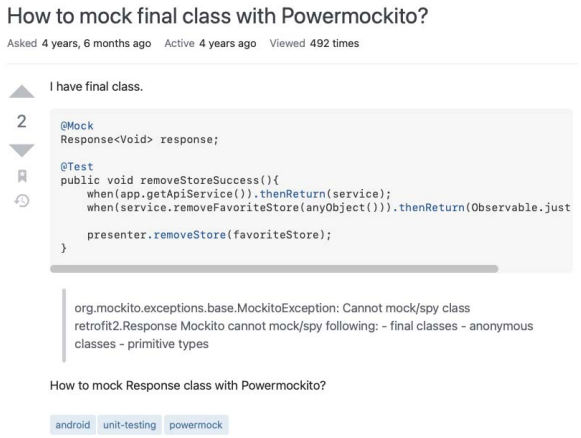

Figure 1: An example in thread 44902324 shows that API *when()* and *thenReturn()* are ambiguous as both *Powermockito* and *Mockito* libraries have APIs with these simple names[1]

Figure 1 shows an issue created by the ambiguity of the simple API names. The context is that an automatic tool is collecting the Stack Overflow threads mentioning the API *when()*, which belongs to the library *Mockito*. While collecting the Stack Overflow threads by searching ones containing the simple API name, the tool cannot find the fully qualified name of those APIs as the code snippet is not complete. *Powermockito* is a library that depends on *Mockito*, so the error log given in the thread is raised by the invocation from *Powermockito*'s API. However, for automatic tools or novice developers, they can be confused by the ambiguity and consider that this thread is about APIs from the *Mockito* library. This may lead to the result that the tool collects incorrect information and it would take more time to clean the collected data. In case a developer

[1]https://stackoverflow.com/questions/44902324

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527918

mistakenly applies the usage of another API, he might get confused because of the incorrect usage and unnecessarily waste his time or even introduce a bug. Therefore, there is a need for an approach that can link the resources/threads containing the ambiguous API mentions to the correct API.

Luong et al. recently proposed DATYS [27], which uses type-scoping to disambiguate API mentions in informal text content on Stack Overflow. In type scoping, they considered API methods whose types appear in more parts (i.e., scopes) of a Stack Overflow thread as more likely to refer to the searched API method. However, the statistical word alignment model it uses is based on the appearance of words in a sentence rather than considering in which context the sequence of words are being used and what connotations do these words relay to the readers.

To incorporate a deeper understanding of the underlying semantics in a natural language text content of Stack Overflow, we introduce ARSeek, a context-specific algorithm to capture the semantic and syntactic information of the paragraphs and code snippets in a crowd-sourced discussion. We call this API resource retrieval task because ARSeek focuses on finding Stack Overflow threads mentioning a given API method. Our work also modified DATYS to perform a better search over the code snippets in the Stack Overflow discussion threads. The modified DATYS, denoted as DATYS+ provides an additional metric to better capture the occurrence of an API method type (i.e., class or interface) in the Stack Overflow thread. By greedily matching the type name with the tokens in the code snippet, DATYS+ performs the syntactic search in ARSeek. Yet, both DATYS and DATYS+ are only searching based on the syntactic information provided by the fully qualified name of a target API method. It cannot capture the semantic meaning in paragraphs and code snippets of threads on Stack Overflow and how similar they are to the target API method. Thus, to capture the semantics, in addition to the weighted syntactic information provided by DATYS+, ARSeek has a semantic search component that leverages a deep attention-based Transformer model, CodeBERT [14]. This semantic search component measures the similarity between the paragraphs and code snippets of a Stack Overflow thread with the target API method comment and implementation code. The more similar they are, the more likely the thread is to be relevant to the target API method that we search for. To efficiently leverage both semantic and syntactic knowledge of the Stack Overflow thread and the API method, ARSeek joins the semantic and the syntactic search element to get the joint relevance score that determines whether a thread relates to a given API method. The contributions of each element in the joint relevance score are defined by a weighting factor.

The significance of this work is that multiple types of information (code snippets, informal textual content) related to API methods are collected to identify relevant APIs for the queries. The collected resources (e.g., thread) from Stack Overflow are a useful knowledge base not only for studies regarding APIs such as API recommendation but also software development in general. Considering that the Maven Central Repository has over 3 million unique software libraries [7], developers can use some help in selecting the right ones for their tasks. To be able to give correct recommendations, the information (e.g., usage, pros, cons, relevant libraries/APIs) related to the libraries or APIs are required. Unfortunately, finding such information for a specific API, which may be buried in the mass of online discussions, is not trivial [55]. With ARSeek, since a large set of related resources (e.g., Stack Overflow threads) from various APIs is collected, developers can directly access the relevant API resources without scouring through the irrelevant ones.

In this paper, we are going to answer the following research questions:

RQ1 Can ARSeek perform better than the baseline (DATYS)?
RQ2 How well does each component of ARSeek perform?
RQ3 How does the weighting factor affect the F1-score of ARSeek?

These research questions will help us understand the effectiveness of our approach ARSeek and the internal mechanism through which it yields better results than the current baseline. Our work has offered the following main contributions:

(1) To our knowledge, we are the first to adopt a transformer-based deep learning technique to incorporate semantic knowledge understanding for API resource retrieval task.
(2) As compared to state-of-the-art techniques, our approach performs better while searching for the contents related to the queried API. On a dataset of 380 Stack Overflow threads, ARSeek beats the state-of-the-art by 14%.
(3) We have also open-sourced our code and additional artifacts required for recreating the results and re-purposing our approach for other tasks. The source code of ARSeek is available at https://github.com/soarsmu/ARSeek.

The rest of the paper is structured as follows: Section 2 deals with the preliminary knowledge about the components on top of which we have built our method. Section 3 provides an overview of our proposed approach, while Section 4 elaborates the various components of our proposed approach. We describe our experiment details and results in Section 5. The related works and the threats to validity are presented in Sections 7 and 6.3. Finally, we concluded our work and present future work in Section 8.

2 PRELIMINARIES

2.1 DATYS

On Stack Overflow, APIs are mentioned informally and these mentions are often ambiguous. When seeking the discussions regarding a specific API, this ambiguity impedes the identification and collection of the correct piece of content mentioning the API. To resolve the ambiguity of the API mentions, API mentions disambiguation links the API mentions with the APIs they refer to. Given identified mentions, DATYS [27] disambiguates Java API mentions via type scoping.

After extracting API method candidates from input Java libraries, DATYS scores API method candidates based on how often their types (i.e., classes or interfaces) appear in different parts (i.e., scopes) of the Stack Overflow thread with identified API mentions. Having a type that appears in more scopes will increase the API candidate score. Here, DATYS considers three scopes: *Mention scope*, which covers the mention itself. *Text scope*, which covers the textual content of the thread, including the mentions. *Code scope*, which covers the code lines inside code snippets in the thread. API candidates are ranked according to their scores for each API mention in the thread. DATYS takes the top API candidate with a non-zero score

as the mentioned API. If the leading API candidate has a zero score, DATYS considers the mention as an unknown API. Luong *et al.* built a ground truth dataset containing 807 Java API mentions from 380 threads in Stack Overflow.

2.2 CodeBERT

CodeBERT [14] was developed using a multilayered attention-based Transformer model, BERT [12]. As a result of its effectiveness in learning contextual representation from massive unlabeled text with self-supervised objectives, the BERT model has been adopted widely to develop large pre-trained models. Thanks to the multi-layer Transformer [52], CodeBERT developers adopted two different approaches than BERT to learn semantic connections between Natural Language (NL) - Programming Language (PL) more effectively.

Firstly, The CodeBERT developers make use of both bimodal instances of NL-PL pairs (i.e., code snippets and function-level comments or documentations) and a large amount of available unimodal codes. In addition, the developers have pre-trained CodeBERT using a hybrid objective function, which includes masked language modeling [12] and replaced token detection [10]. The incorporation of unimodal codes helps the replaced token detection task, which in turn produces better natural language by detecting plausible alternatives sampled from generators.

Developers trained CodeBERT from Github code repositories in 6 programming languages, where only one pre-trained model is learned for all six programming languages with no explicit indicators used to mark an instance to the one out of six input programming languages. CodeBERT was evaluated on two downstream tasks: natural language code search and code documentation generation. The study found that fine-tuning the parameters of CodeBERT obtained state-of-the-art results on both tasks.

3 APPROACH OVERVIEW

3.1 Task Definition

Our goal is to find Stack Overflow threads that mention a given API method[2]. Specifically, given an API method, we strive to find Stack Overflow threads containing words matching the simple name of the given API method. In Java, the simple name of an API method is the name of the method without the class and the package names. For example, `m` is the simple name of API method `com.example.Class.m`. We want to classify whether the thread having the simple name `m` is actually relevant to the API method `com.example.Class.m`. In summary, the task is defined as: *"For each API method in a set of given API methods, identify Stack Overflow threads that refer to it."*

3.2 Architecture

The pipeline of ARSeek is presented in Figure 2. It is divided into 2 main steps:

(1) Collecting various API-related resources from a given API method name; and (2) Recommending relevant threads using the collected API-related resources.

[2]In this paper, we use the terms API, method, and API method interchangeably.

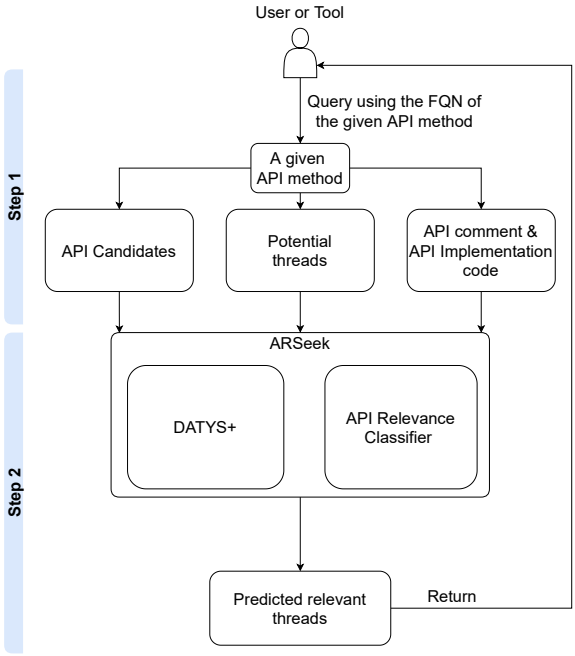

Figure 2: The architecture of ARSeek

In step (1), ARSeek finds *Potential Threads* from Stack Overflow using the simple name of the given API method as the query. *Potential Threads* are the threads that have at least one word matching with the simple name of the given API. The *API method comment and implementation code* are directly obtained from the source code repository of the given API. Last but not least, the *API Candidates* are obtained from a database of API methods as a knowledge base as same as [27, 56]. The *API Candidates* are API methods that have the same simple name as the given API method.

The objective of step (2) is to identify whether each Stack Overflow thread in the *Potential Threads* actually refers to the given API. ARSeek has two components: *API relevance classifier* and DATYS+. *API relevance classifier* is designed to draw the relevance between a thread and an API method by capturing the semantic similarity between (1) paragraphs and code snippets in the thread; and (2) API method comment and implementation code. *API relevance classifier* outputs a semantic relevance score representing the relevance it measures. In contrast, DATYS+ outputs a syntactic relevance score based on the existence of the terms from the fully qualified name of the given API in different scopes of a Stack Overflow thread. For example, API "A.B.c" has terms such as "A", "B", and "c". The last term, "c", is the simple name of the API method. The second last term, "B", is the type of the API method. Both DATYS and DATYS+ use type scoping [27] to give a score based on the existence of the type of the API (i.e., "B" in the example) in different scopes of the Stack Overflow thread (code scope, text scope, etc.). A new scope of DATYS+ is modified to be suitable for the search task and we are going to describe it in Section 4.1. It outputs a score that indicates the syntactic relevance between the given API and the thread, we call it DATYS+ score.

After step (2), each thread will have a score indicating if the thread refers to the given API method. This score is combined from semantic relevance score and DATYS+ score and is called *joint relevance score*. Threads predicted referring to the given API method are then returned to the user. We describe ARSeek components (i.e., DATYS+ and *API relevance classifier*) in detail in Section 4.

4 ARSEEK

ARSeek consists of two main components: DATYS+ and *API relevance classifier*. DATYS+ takes as inputs *Potential Threads* and *API Candidates* and outputs scores indicating its confidence that the given API is referred to in the threads (Section 4.1). Given *Potential Threads* and *API method comment and implementation code*, ARSeek first converts them to *API relevance embedding* (Section 4.2). The *API relevance embedding* is input to *API relevance classifier*, which outputs confidence scores indicating the likelihood that the given threads refer to the API (Section 4.3). Finally, the scores from DATYS+ and *API relevance classifier* are combined to a joint relevance score, and threads with scores larger than a threshold are returned as the relevant threads (Section 4.4).

4.1 DATYS+

DATYS+ is an extension of DATYS. DATYS used regular expressions to capture the types of API method invocations available in code snippets of the thread. However, these regular expressions are limited and thus DATYS may miss some mentions in code snippets. To capture more types, DATYS+ modifies the type scoping algorithm by adding a new score. This score helps capture the occurrence of types that would be missed by DATYS scopes. To be specific, DATYS has *Code Scope* and *Text Scope* to capture method types available in the code snippets and textual content of the thread, respectively. However, when a method type is available in a comment inside a code snippet, DATYS cannot capture it as it neither belongs to *Code Scope* nor *Text Scope* of DATYS. Thus, DATYS+ adds code comments to the *Code Scope* and refers to the new scope as the *Extended Code Scope*, highlighting the extension made by DATYS+ over DATYS.

Algorithm 1 indicates how modified type scoping works. Compared to DATYS's, DATYS+'s type scoping algorithm receives *CodeComment* as input. *CodeComment* represents the comments inside code snippets of the Stack Overflow thread. In addition, inputs of the original type scoping algorithm are also considered. *APIMention*, *PTypeList*, *APIMethodCandidate*, and *ThreadContent* stand for the simple name of the given API, the list of possible types extracted from code snippets following the algorithm used by DATYS, the *API Candidates*, and the thread's textual content (i.e., title, text, tags), respectively. The three scopes used by DATYS are also used in DATYS+. In *Mention Scope (Lines 3-8)*, DATYS+ increases an API score if its type appears within the API mention. In *Text Scope (Lines 10-13)*, DATYS+ increases an API score if its type appears within the textual content of the thread. In *Code Scope (Lines 14-18)*, DATYS+ increases an API score if its type matches with the type of method invocations or imported types in the code snippet. Additionally, in *Extended Code Scope*, DATYS+ also looks at the content of the comments inside the code snippets and increases the API score of the corresponding API candidate if there are tokens

Algorithm 1 Scoring an API Candidate with Type Scoping in DATYS+

```
Input: ApiMention, PTypesList, APIMethodCandidate,
       ThreadContent, CodeComment
Output: CandScore
 1: CandScore = 0
 2: CandType = getType(APIMethodCandidate)
 3: if hasPrefix(ApiMention) then
 4:     Prefix = getPrefix(ApiMention)
 5:     if endsWith(Prefix, CandType) then
 6:         CandScore = CandScore + 1
 7:     end if
 8: end if
 9: TextualTokens = tokenize(ThreadContent)
10: CommentTokens = tokenize(CodeComment)
11: if CandType in TextualTokens then
12:     CandScore = CandScore + 1
13: end if
14: for PType in PTypesList do
15:     if isSameType(PType, CandType) then
16:         CandScore = CandScore + 1
17:     end if
18: end for
19: if CandType in CommentTokens then
20:     CandScore = CandScore + 1
21: end if
22: return CandScore
```

in the code snippets or comments that match with the API type (Lines 19-21).

After executing type scoping, DATYS+ returns scores for the *API Candidates*. The scores are then normalized to a range of [0, 1] following the minimum and the maximum score from the *API Candidates*. DATYS+ then takes the normalized score of the given API method and passes it to the next step.

4.2 *API relevance embedding*

We follow the process described in Figure 3 to build *API relevance embedding*. Firstly, each thread in *Potential Threads* needs to be converted into an embedding. We define a thread as a question post and its answer posts. Comments of posts are excluded, as was also done in the DATYS paper [27]. A thread may contain m paragraphs and n code snippets. A paragraph is a piece of textual content on a Stack Overflow thread that is separated from other contents in the thread via a newline character. A code snippet is a piece of code content on a Stack Overflow thread. It is typically enclosed with a starting tag $\langle pre \rangle \langle code \rangle$ and an ending tag $\langle /code \rangle \langle /pre \rangle$. To be more specific, the $\langle code \rangle$ $\langle /code \rangle$ tags covered by the $\langle pre \rangle$ $\langle /pre \rangle$ tag represent the code snippets while the ones without the $\langle pre \rangle$ $\langle /pre \rangle$ tag are highlighted words/sentences within the paragraphs. Each paragraph is paired with each code snippet to create a pair of *thread content*. Therefore, a Stack Overflow thread would have $m \times n$ *thread content* pairs. A natural-programming language model, CodeBERT[3], is used to extract the semantic meaning of each *thread content* pair. It encodes the $m \times n$ *thread content* pairs into $m \times n$ *thread embeddings*. *thread embedding* is the representation vector of *thread content* created by CodeBERT's encoder. By converting the pairs from a textual form to a numerical vector form with a pre-trained CodeBERT model, the semantic relationship between the paragraphs and code snippets is extracted. Before feeding the *thread content* pairs into the encoder of CodeBERT, each pair is

[3]https://github.com/microsoft/CodeBERT

pre-processed following the format:

$$\langle CLS \rangle\ paragraph\ \langle SEP \rangle\ code\ snippet\ \langle EOS \rangle$$

$\langle CLS \rangle$ is the token that informs the start of the pair according to the design of RoBERTa model [26] which CodeBERT is based on. $\langle SEP \rangle$ is the token that separates a Paragraph from a Code Snippet and $\langle EOS \rangle$ indicates the end of the pair.

The maximum number of tokens in a pair before being fed into the CodeBERT encoder is 512, which is twice the number of tokens used in the original CodeBERT [14]. This is intended to provide more contextual information to the CodeBERT encoder. We consider the paragraph and the code snippet to be equally important. Therefore, each of them is given 256 tokens. However, we have to cater the $\langle CLS \rangle$, $\langle SEP \rangle$, and $\langle EOS \rangle$ tokens. Thus, the number of tokens for a paragraph and a code snippet is set to 254 and 255 tokens, respectively. The two numbers add up to 512 when the three tokens are counted. If the number of tokens in the paragraph is less than 254, then padding tokens would be added to reach 254 tokens. On the other hand, if the number of tokens in the paragraph is more than 254, we truncate the paragraph following [26] and take the first 254 tokens. The same process is applied to the code snippet with 255 tokens. The CodeBERT encoder receives these *thread content* pairs under this format as inputs and outputs embedding vectors. For a thread with $m \times n$ *thread content* pairs, there would be $m \times n$ *thread embedding* vectors created and each *thread embedding* vector has a length of 768. We created $m \times n$ embeddings for *thread content* pairs rather than one embedding since the content of a whole thread cannot be fit into CodeBERT at once (i.e., due to the limited number of input tokens of the CodeBERT's encoder) and we want to capture all the information from the thread.

Secondly, to build *API relevance embedding*, *API comment and implementation code* also need to be converted into an embedding. The API method comment (i.e., Javadoc) is a piece of textual content that describes the functionality of the API method and how to use it. The API implementation code is the code inside the API method body that implements the described functionality. JavaParser is used to extract the *API comment and implementation code* from the Javadoc and the JAR files, respectively. They are pre-processed to the following format:

$$\langle CLS \rangle\ comment\ \langle SEP \rangle\ implementation\ code\ \langle EOS \rangle$$

They are then transformed into a numerical representation vector via the CodeBERT encoder.

Finally, each *thread embedding* vector and the *method embedding* vector are then concatenated to a vector. We call this concatenated vector *API relevance embedding*. In total, $m \times n$ *API relevance embedding* vectors would be created.

4.3 *API relevance classifier*

The *API relevance classifier* is a binary classifier that utilizes a neural network with two fully connected layers to predict whether the *API relevance embedding* comes from a Stack Overflow thread that refers to the given API method.

The *API relevance classifier* has two modes of operation: training and deployment modes. In the training mode, the *API relevance embeddings* are used to train the *API relevance classifier*. When there is an imbalance between positive and negative labels, the *API relevance classifier* upsamples the minority label. Whenever the thread refers to the given API method, all *API relevance embedding* created from the thread would be considered as positive by the classifier. Otherwise, in case the given API method is not referred to by the thread, every *API relevance embedding* of the thread would have negative labels. In the deployment mode, the *API relevance classifier* produces probability scores for the $m \times n$ *API relevance embedding*. These scores are averaged and passed to the next step. The averaged score indicates the likelihood that the thread refers to the given API.

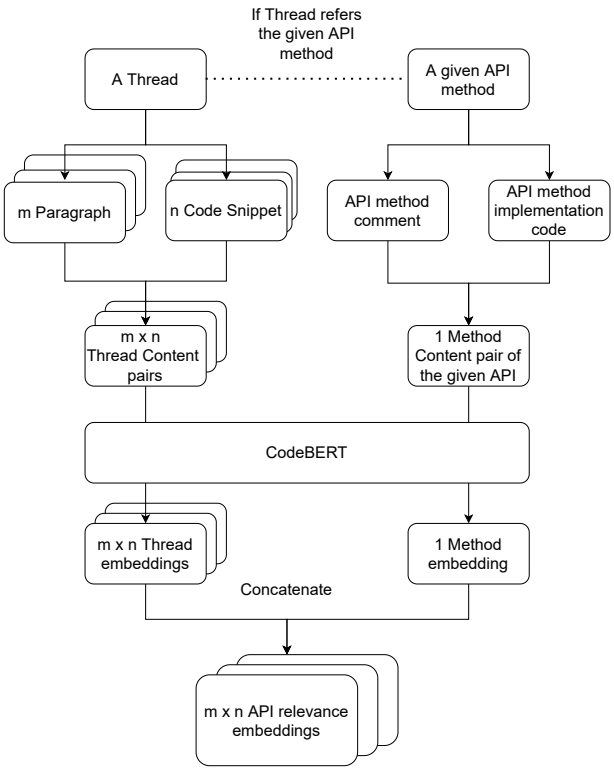

Figure 3: How *API relevance embeddings* are created

4.4 Computing joint relevance score

We follow the process in Figure 4 to compute the joint relevance score. DATYS+ and *API relevance classifier* output scores A and B, respectively. Both represent their confidence that the given API method is mentioned in the thread. While A represents the score given by the syntactic information of DATYS+, B is given by the semantic information from *API relevance classifier*. The two scores are then combined to a joint relevance score C following this formula:

$$C = x \times A + (1 - x) \times B \tag{1}$$

The weighting factor x decides the contributions of DATYS+ score and *API relevance classifier* in joint relevance score C. The higher the value of x is, the more DATYS+ score contributes to the final joint relevance score. The range of A, B, and x is from 0 to 1. A thread is considered to refer to the given API if the joint relevance score C is

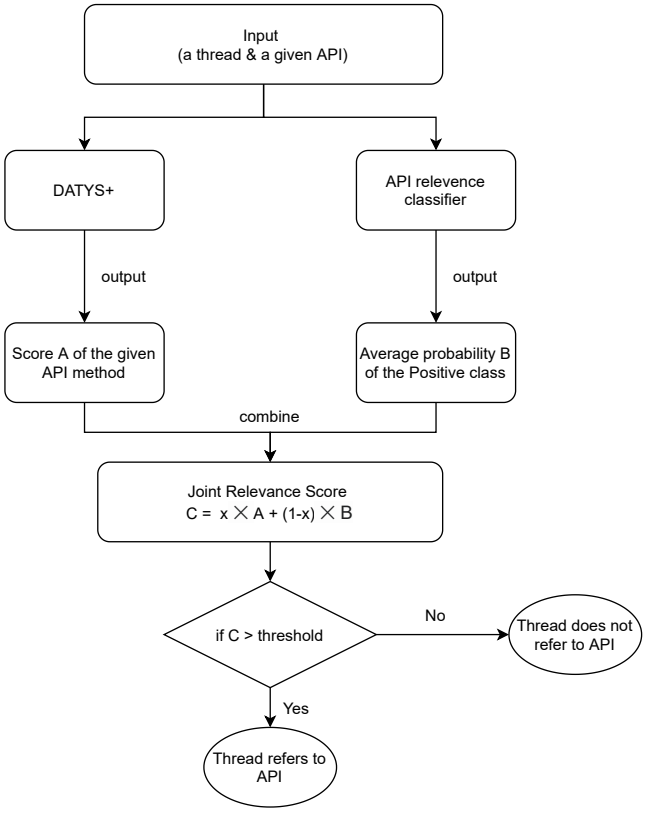

Figure 4: Computing joint relevance score

larger than a threshold t. Otherwise, the thread is considered not to refer to the given API. By default, t is set to 0.5.

The value of x will be estimated based on the training data. In detail, we let x increase gradually from 0 to 1 with a step of 0.1. There are ten possible values of x: $\{0, 0.1, 0.2, ..., 0.9, 1.0\}$. The value of x giving the highest performance on the training data is then chosen.

5 EXPERIMENT RESULT

5.1 Dataset and Experimental Settings

We utilize the dataset provided in DATYS work [27] to evaluate both ARSeek and DATYS. We split 380 Stack Overflow threads into training threads and testing threads using 3-fold cross-validation. The training threads are utilized to train the *API relevance classifier* while the testing threads are used to evaluate ARSeek and the baseline (i.e., DATYS). Next, as mentioned in Section 4.2, for each Stack Overflow thread in the training threads, we extract its *thread embedding*s and these *thread embedding*s are grouped into a training set. Similarly, for each Stack Overflow thread in the testing threads, we extract its *thread embedding*s and these *thread embedding*s are grouped into a testing set. The number of *API relevance embedding*s of each fold on the dataset is shown in Table 1. As the embeddings are created from the combination of paragraphs and code snippets in each Stack Overflow thread, the numbers of the embeddings on training sets of fold 1, 2, and 3 are 57 690, 59 660, 58 017, respectively, while the numbers on testing sets of fold 1, 2, and 3 are 26 212, 24 242, and 25 885, respectively.

Table 1: Number of *API relevance embedding*s in each set

	Fold 1	***Fold 2***	***Fold 3***
Training set	57,690	59,660	58,017
Testing set	26,212	24,242	25,885

Table 2: Number of positive and negative *API relevance embedding*s in each set

	Fold 1	***Fold 2***	***Fold 3***
Positive Embeddings in Training set	9,934	10,878	10,131
Negative Embeddings in Training set	47,756	48,782	47,886
Positive Embeddings in Testing set	5,607	4,663	5,410
Negative Embeddings in Testing set	20,605	19,579	20,475

To generate *API relevance embedding*s for the *API relevance classifier* for training, for each thread, if the given API appears in the thread, we generate *API relevance embedding*s for *thread content*s and *method content*s as described in Section 4.2. These embeddings would have a positive label because they are created from the API that is referred to by the thread. To generate embeddings with a negative label for a thread, we find APIs that have the same simple name as the given API and are not mentioned in the thread. We then create *API relevance embedding*s from these APIs and label these *API relevance embedding*s as negative.

While the APIs on the training set are used to generate the training *method embedding*s. The testing *method embedding*s on the corresponding testing set are created from the APIs on that testing set. Table 2 shows the numbers of positive and negative *API relevance embedding*s created on training and testing sets. The negative *API relevance embedding*s for an API method are collected from threads that do not refer to the API method while the positive ones are collected from threads referring to the API method. Since the number of threads that do not refer to the API method is higher than the number of threads that refer to it, the number of negative *API relevance embedding*s is naturally higher than the number of positive ones. The number of negative *API relevance embedding*s is approximately four times the number of positive ones on the set of threads in our dataset. Due to this imbalance, positive *API relevance embedding*s are randomly up-sampled to balance the two classes within the *API relevance classifier* training process. The *API relevance classifier* is trained using 6 epochs and the learning rate of 10^{-3}. After the first 6 epochs, the value of the loss function has relatively converged.

5.2 Metrics

To evaluate the proposed approach on identifying threads that are relevant to an API, we use three metrics: Precision, Recall, and F_1-*score*. In order to calculate the three aforementioned metrics, True Positive, False Positive, and False Negative should be defined first. Our task focuses on finding threads that actually refer to a given API. True Positive is the case where a thread is deemed to

be relevant by the approach is indeed relevant. False Positive is the case where the thread that is deemed to be relevant by the approach is actually irrelevant. False Negative is the case where a thread that is deemed to be irrelevant by the approach is relevant. The metrics are calculated using the following formulas:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (2)$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (3)$$

$$F_1\text{-}score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

We measure the above scores of all given APIs in the testing set and report the averages of the scores.

5.3 Research Questions

Research Question 1: Can ARSeek perform better than the baseline (DATYS)?
The baseline, DATYS, was designed for a task of API mention disambiguation. We adopt it to our task of finding threads that are relevant to an API. If DATYS finds an API is mentioned in the thread, the thread is considered to be relevant to the API. To evaluate the improvement over DATYS, we evaluate them on the testing data set and compare them in terms of F_1-$score$. We also analyze some cases that ARSeek can resolve and DATYS cannot in Section 6.1.

Research Question 2: How well does each component of ARSeek perform?
There are three possible variants of ARSeek depending on which component that comes along with it. The variants are (1) ARSeek with *API relevance classifier*; (2) ARSeek with DATYS+; and (3) ARSeek with DATYS+ and *API relevance classifier*. *API relevance classifier* is a semantics-based algorithm while DATYS+ is a syntactic-based algorithm. In this study, we aim to analyze the contribution of each component to ARSeek. From the analysis, we would like to answer the question of whether combining a semantic-based algorithm and a syntactic-based algorithm leads to a better result than running them individually.

Research Question 3: How does the weighting factor affect the F1-score of the relevant thread classification? Does our strategy work well?
The weighting factor is an important factor that would affect how well ARSeek performs. We select the importance factor based on the best performance on the training data. We analyze whether our strategy leads to the best performance on the testing data. We vary the values of the weighting factor in both the training data and the testing data. The values that we use are $\{0, 0.1, 0.2, ..., 0.9, 1.0\}$. We analyze whether picking values in the training data that leads to the best performance on the training data also leads to the best performance on the testing data.

5.4 RQ1: ARSeek Effectiveness

Table 3 shows the performance of the DATYS and ARSeek in finding threads that are relevant to the given API. ARSeek, in general, outperforms DATYS. On average, ARSeek achieves an F1-score of 0.8709, which is an improvement of 14% compared to DATYS. ARSeek also beats DATYS in terms of precision and recall. As DATYS only focuses on the syntactic information available in a Stack Overflow thread, it returns incorrect results when the scopes (Code Scope, Text Scope, Mention Scope) that it tracks do not contain the API method name. DATYS+ improves the result by adding the *Extended Code Scope*. By combining DATYS+ scope with the semantic information from the *API relevance classifier*, ARSeek avoids picking irrelevant threads that are within the scope.

Table 3: ARSeek vs DATYS in terms of average Precision, Recall, and F_1-score on the testing set in 3-fold cross validation

Approach	Avg. Precision	Avg. Recall	Avg. F_1-score
DATYS	0.7372	0.7718	0.7304
ARSeek	0.8725	0.8962	0.8709

Table 4: Contribution of ARSeek Components

Components	Avg. Precision	Avg. Recall	Avg. F_1-score
ARSeek (1)	0.8725	0.8962	0.8709
ARSeek with only API relevance classifier (2)	0.3792	0.4137	0.3821
ARSeek with only DATYS+ (3)	0.8673	0.8795	0.8596

Table 5: Average Precision, Recall, F_1-score on testing sets in 3-fold cross validation when weighting factor varies

x	Avg. Precision	Avg. Recall	Avg. F_1-score
0	0.3792	0.4137	0.3821
0.1	0.4943	0.5187	0.4943
0.2	0.7795	0.8039	0.7770
0.3	0.8497	0.8800	0.8515
0.4	**0.8725**	**0.8962**	**0.8709**
0.5	0.8612	0.8782	0.8416
0.6	0.8631	0.8802	0.8573
0.7	0.8637	0.8802	0.8577
0.8	0.8637	0.8802	0.8577
0.9	0.8637	0.8802	0.8577
1.0	0.8673	0.8795	0.8596

5.5 RQ2: Ablation Study

Table 4 shows how well each component in ARSeek is. We can see that the highest average F_1-$score$, which is 0.8709, is achieved by "ARSeek"(1) when weighting factor of x is set to 0.4. The F_1-$score$ of "ARSeek with only *API relevance classifier*"(2) and "ARSeek with only DATYS+"(3) are 0.3821 and 0.8596, respectively. Since the "ARSeek with only *API relevance classifier*" version gives the worst result, the *API relevance classifier* may not be able to resolve the task well independently. Partly, this might be due to the limited amount of the training data (i.e., only 253 training threads). In addition, the "with

only DATYS+" version of ARSeek performs much better compared to the "ARSeek with only *API relevance classifier*" version. As the difference between the ARSeek (1) and "ARSeek with only DATYS+" (3) is small in terms of average F_1-$score$, we perform the statistical test to examine whether there is a statistically significant difference between the result (i.e., F_1-$score$) of ARSeek and "ARSeek with only DATYS+". In detail, Mann-Whitney U test [30] is used and the result is significant at the 0.05 level. Thus, the semantic component *API relevance classifier* helps improve the syntactic component DATYS+.

5.6 RQ3: Effect of the weighting factor

Table 5 shows the performance of ARSeek on the test set when we vary the value of the weighting factor x. The bold numbers in each column of the table are the results of the chosen weighting factor from the training sets. To be specific, we vary the weighting factor x on the training sets. Then we choose the value that results in the highest average F_1-$score$ of the 3 training sets. This chosen weighting factor is equal to 0.4 and it is used for the model to make decisions on the testing sets. From Table 5, the results show us the best combination is also achieved when the weighting factor is 0.4. In detail, the average F_1-$score$ of the 3 folds increases and reaches its peak when the weighting factor is 0.4. After that, the results slightly drop when the weighting factor increases to 1.

6 DISCUSSION

6.1 Cases where ARSeek outperforms DATYS

(1) *The relevant thread does not contain the type name of the given API method*

Figures 5 and 6a show an example of a case where the content of the thread does not relate to the given API method. As a Stack Overflow thread may contain one question post and multiple answer posts, Figure 6a represents the question post while the answer post is shown in figure 5. Figure 5 contains a paragraph and code snippet of a post belonging to thread 56135373[4]. The fully qualified name *org.mockito.stubbing.OngoingStubbing.thenReturn*[5] is the API method the thread refers to.

From the content of the thread in Figures 5 and 6a, it would be difficult to find the relevance between the text written in the paragraphs and the given API method (i.e., *org.mockito.stubbing.OngoingStubbing.thenReturn*) since the type (i.e., *OngoingStubbing*) does not appear in the thread. The text in figure 5 just shows the user view towards the code snippet without having a description mentioning the application or usage of the observed API method invocation (e.g., *thenReturn* in the code snippet of Figure 5). Sentences such as *"This works like charm!"* do not provide much information to identify whether the observed API method refers to the given API.

Therefore, we leverage the content of the thread which might be relevant to the content of the API method. For example, in the thread above, its title which is shown in Figure 6a, *"Optional cannot be returned by stream() in Mockito Test classes"*, relates to the comment of the given API which is *Sets a return value to be returned when the method is called* in Figure 6b. Due to this similar relation, ARSeek can successfully consider this thread as relevant while DATYS missed it.

Figure 5: Thread 56135373 on Stack Overflow where API is referred by a code snippet of the thread

(2) *The irrelevant thread contains the type name of the given API method*

An example of this case is shown in Figure 7. In the thread[6], the given API method is *com.google.common.base.CharMatcher.is* and there is a word that matches the simple name of the API method *is* which we highlighted. Since the type of the given API method (e.g., *CharMatcher*) appears in both the textual content and the code snippet, DATYS mistakenly accepts the thread as referring to the given API. By leveraging the semantic knowledge learned by the *API relevance classifier*, ARSeek is able to detect the irrelevance between the textual content, code snippet around the word *is* and the API comment and implementation code. ARSeek can conclude that the thread is irrelevant to the given API *com.google.common.base.CharMatcher.is*.

6.2 Example case where ARSeek fail to exclude irrelevant threads

Figure 8 shows a case where ARSeek fails to exclude the thread[7] out of the relevant results for the given API method *org.mockito.Mockito.mock*. The issue occurs when there is an API method that has similar functionality as the given API method. These two methods usually have the same simple name and highly similar description.

In Figure 8, *PowerMock* and *Mockito*, perform similar functions such as mocking (i.e., creating a version of a service in order to quickly and reliably run tests on that service[8]). Since both of them have the API method whose simple name is *mock*, and both of their *mock* methods have the same API signature (i.e., parameters, return type), it would be easy to mistakenly recognize one as the other. Figure 9 shows the comment of API method *org.mockito.Mockito.mock*[9], which is *Creates mock object of the given class or interface*. Because of the similarity between the API method from *Mockito* library and the title of thread 30127057 in Figure 8, ARSeek wrongly recognizes that the simple API name *mock* in the thread refers to the given API method *org.mockito.Mockito.mock*. In fact, the simple API name *mock* refers to the one from the *PowerMock* library.

[4]https://stackoverflow.com/questions/56135373/
[5]https://javadoc.io/doc/org.mockito/mockito-all/2.0.2-beta/org/mockito/stubbing/OngoingStubbing.html
[6]https://stackoverflow.com/questions/16919751/
[7]https://stackoverflow.com/questions/30127057/
[8]https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/
[9]https://javadoc.io/static/org.mockito/mockito-all/2.0.2-beta/org/mockito/Mockito.html

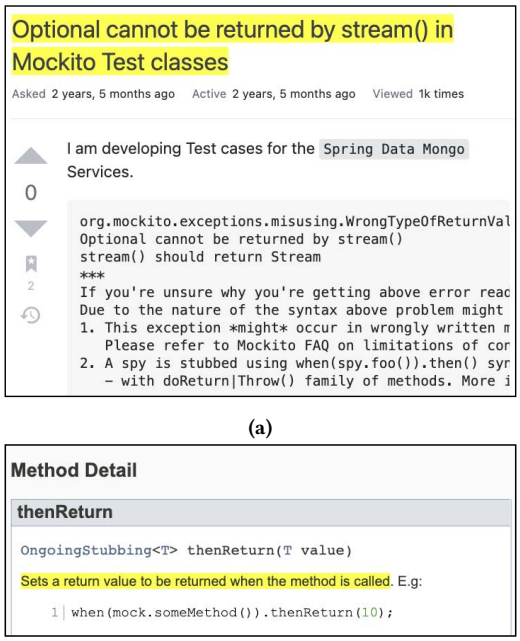

Figure 6: The similarity in semantic meaning between the API comment of method *org.mockito.stubbing.OngoingStubbing.thenReturn* in Figure 6a and the textual content (i.e., the title) of thread 56135373 in Figure 6b

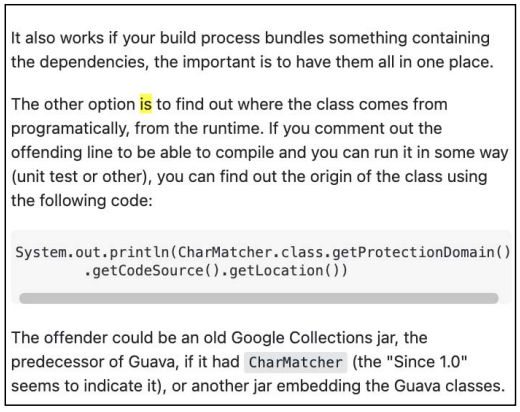

Figure 7: Thread 16919751 on Stack Overflow where API *com.google.common.base.CharMatcher.is* is not referred by the content of the thread.

6.3 Threats to validity

A threat to internal validity is related to errors in our code base. We check our code multiple times but there still could be errors that we did not notice.

A threat to external validity is related to the generalizability of our approach. This experiment mainly focuses on analyzing Stack Overflow threads in Java programming language, therefore, it is uncertain whether ARSeek can be applied to other discussion

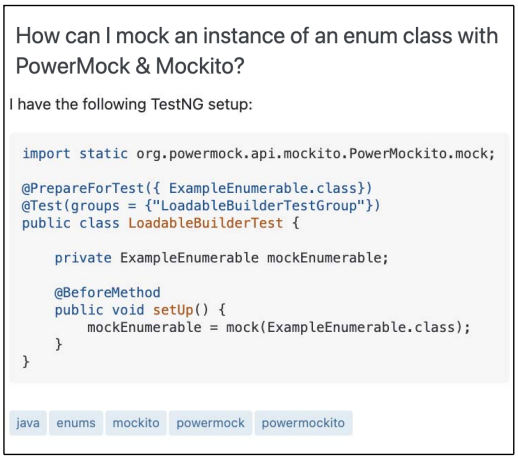

Figure 8: Thread 30127057 on Stack Overflow that ARSeek falsely recognize as referring to the API method *org.mockito.Mockito.mock*.

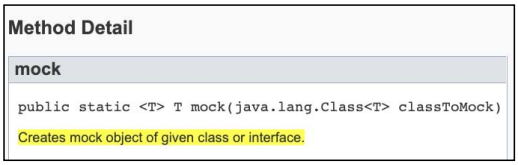

Figure 9: The API comment of method *org.mockito.Mockito.mock*

venues that also talk about API issues. We pick Stack Overflow and Java as they are one of the most popular Software Question and Answer (SQA) sites and programming languages, respectively. Indeed, changing the target programming language could affect the accuracy of ARSeek. However, although we focus on Java, the features (e.g., API comment/documentation, API implementation code, fully qualified name, class/type name, etc.) required for the approach can be found in other programming languages.

A threat to construct validity in concerned on whether precision, recall, and F_1-*score* is a suitable evaluation metric. Our task is a classification task and a lot classification work in software engineering have used precision, recall, and F1-score as the evaluation metric [18, 27, 38, 56]. Thus, we believe the threat is minimal.

7 RELATED WORK

API Disambiguation. Many publications [3, 4, 11, 31, 37, 43, 47, 56] deal with API disambiguation. There are two main groups: informal text disambiguation [3, 4, 11, 31, 56] and code snippet disambiguation [37, 43, 47]. As suggested by its name, the first aims to disambiguate API mentions in textual content while the second deals with disambiguating API mention in code snippets.

For informal text disambiguation, several works utilize classical information retrieval approaches such as Vector Space Model and Latent Semantic Indexing to disambiguate the API mentions [3, 4, 31] while some others use heuristics [11]. Bacchelli et al [4]

combined string matching and information retrieval algorithms to link emails to source code entities. Dagenais and Robillard [11] identified Java APIs mentioned in support channels (e.g., mailing list, forums), documents, and code snippets. Ye *et al.* [56] worked on API disambiguation in the textual content of Stack Overflow thread by utilizing mention-mention similarity, mention-entry similarity, and scope filter. Luong et al.[27] used type scoping to disambiguate the API mentions in a Stack Overflow thread.

The work on API disambiguation on Stack Overflow thread can be viewed as another side of the coin of the task in finding threads that are relevant to the API. When we disambiguate an API in a thread, the disambiguated API is relevant to the thread as the thread is talking about the API.

API Resource Retrieval. Several studies have explored how to search for the code for API and related information retrieval. Lv et al. [28] proposed Codehow to deal with the lack of query understanding ability of the existing tool. By expanding a user query with APIs, Codehow can identify potential APIs and perform a code search based on the Extended Boolean model, which considers the impact of APIs on code search. Gu et al. [17] proposed DeepAPI to search for API usage sequences. As opposed to assuming a bag of words, it learns the sequence of words within a query and the sequence of APIs associated with it. DeepAPI encodes a single user query into a fixed-length context vector to generate an API sequence.

Other studies have also exploited different aspects of APIs and natural language to better retrieve the APIs and their related information. The techniques include: using global and local contexts of the queries [34], leveraging usage similarity for effective retrieval of API examples [5], employing word embeddings to document similarities for improved API retrieval [57], exploiting user knowledge [42], and task-API knowledge gap [20] during retrieval of semantically annotated API operations.

Wang et al. [54] developed a transformer-based framework for unifying code summarization and code search. Shahbazi et al. [44] proposed API2Com to improve automatically generated code comments by fetching API documentation. Alhamzeh et al. [2] built DistilBERT-based argumentation retrieval for answering comparative questions. Dibia et al. [13] and Vale et al.[51] developed a usable library for question answering with contextual query expansion and a question-answering assistant for software development using a transformer-based language model, respectively. Ciniselli et al.[9] performed an empirical study on the usage of Transformer Models for code search and completion.

Our study also works on API resource retrieval. Specifically, we retrieve Stack Overflow threads that are relevant to a target API that we are searching for.

API Documentation. Treude et al. [48] studied the augmenting API documentation with insights from stack overflow. [35] explored the Crowd documentation by examining the dynamics of API discussions on Stack Overflow, whereas [45] dubbed the Stack Overflow as the Social Media for Developer Support in terms of provided utilities. [1] and [6] worked on classifying stack overflow posts on API issues and contextual documentation referencing on stack overflow. The dichotomy of these studies is notable where some research like [50] and [58] studies how API documentation fails via the API misuse on stack overflow, other studies [32, 41] heavily lean on the Crowdsourced knowledge on stack overflow for automated API documentation with tutorials. Similarly, crowdsourced knowledge was hailed by [16] and [24], who explored the innovation diffusion and web resource recommendation for hyperlinks through link sharing on stack overflow.

Our work supports the effort in this line of study. ARSeek can automatically find threads about a particular API in Stack Overflow that can be augmented to the corresponding API documentation.

Word Sense and Entity Disambiguation Study. There are several works focused on the disambiguation task [11, 33, 37, 47]. We also have found a variety of word sense and entity disambiguation methods employed for different objectives [8, 15, 21–23, 29, 36, 46, 59]. These studies have solved myriads of problems via solving lexical disambiguations in literature. The task of word sense disambiguation is to identify a target word's intended meaning by examining its context. Researchers have used Word Sense Disambiguation to predict election results by enhanced sentiment analysis on Twitter data. Researchers have associated place-name mentions in unstructured text with their actual references in geographic space using word disambiguation. Other research has also proposed unsupervised, knowledge-Free, and interpretable Word Sense Disambiguation for various applications. Researchers used this approach to add meaning to social network posts when it comes to named entity recognition and disambiguation. Different Entity-fishing tools were also developed for facilitating the recognition and disambiguation service. In recent years, tools that allow researchers to recognize and extract named entities have become increasingly popular.

8 CONCLUSION AND FUTURE WORK

We present ARSeek, an approach to search Stack Overflow threads that refer to API of which users or tools may want to find the usage. We utilize the semantic and syntactic features of the paragraphs and code snippets in a thread to determine whether the thread is related to a given API. Our evaluation shows that ARSeek has an improvement compared to DATYS when adapting both approaches to the search task. We have added a weight parameter to balance the usage of syntactic and semantic information for retrieving API mentions and related threads. We have proved the utility of the weight factor by incorporating an ablation study. In the future, we plan to improve our approach with a larger dataset that has more threads and APIs. Also, we plan to make our approach become robust with more programming languages so that it can be more useful to developers.

Replication Package. The source code for ARSeek is available at https://github.com/soarsmu/ARSeek.

Acknowledgment. This research / project is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award MOE2019-T2-1-193). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. 2018. Classifying stack overflow posts on API issues. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 244–254.

[2] Alaa Alhamzeh, Mohamed Bouhaouel, Előd Egyed-Zsigmond, and Jelena Mitrović. 2021. DistilBERT-based Argumentation Retrieval for Answering Comparative Questions. *Working Notes of CLEF* (2021).

[3] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *TSE* 28, 10 (2002).

[4] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. In *ICSE*.

[5] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/1882291.1882316

[6] Sebastian Baltes, Christoph Treude, and Martin P Robillard. 2020. Contextual documentation referencing on stack overflow. *IEEE Transactions on Software Engineering* (2020).

[7] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 344–348.

[8] Ping Chen, Wei Ding, Chris Bowes, and David Brown. 2009. A fully unsupervised word sense disambiguation method using dependency knowledge. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. 28–36.

[9] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *arXiv preprint arXiv:2108.01585* (2021).

[10] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[11] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *2012 34th international conference on software engineering (icse)*. IEEE, 47–57.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Victor Dibia. 2020. Neuralqa: A usable library for question answering (contextual query expansion+ BERT) on large datasets. *arXiv preprint arXiv:2007.15211* (2020).

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

[15] Luca Foppiano and Laurent Romary. 2020. entity-fishing: a DARIAH entity recognition and disambiguation service. *Journal of the Japanese Association for Digital Humanities* 5, 1 (2020), 22–60.

[16] Carlos Gómez, Brendan Cleary, and Leif Singer. 2013. A study of innovation diffusion through link sharing on stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 81–84.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[18] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[19] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *ASE*. IEEE.

[20] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 293–304. https://doi.org/10.1145/3238147.3238191

[21] Rincy Jose and Varghese S Chooralil. 2015. Prediction of election result by enhanced sentiment analysis on Twitter data using Word Sense Disambiguation. In *2015 International Conference on Control Communication Computing India (ICCC)*. 638–641. https://doi.org/10.1109/ICCC.2015.7432974

[22] Morteza Karimzadeh, Wenyi Huang, Siddhartha Banerjee, Jan Oliver Wallgrün, Frank Hardisty, Scott Pezanowski, Prasenjit Mitra, and Alan M. MacEachren. 2013. GeoTxt: A Web API to Leverage Place References in Text. In *Proceedings of the 7th Workshop on Geographic Information Retrieval* (Orlando, Florida) *(GIR '13)*. Association for Computing Machinery, New York, NY, USA, 72–73. https://doi.org/10.1145/2533888.2533942

[23] Dan Klein, Kristina Toutanova, H Tolga Ilhan, Sepandar D Kamvar, and Christopher D Manning. 2002. Combining heterogeneous classifiers for word sense disambiguation. In *Proceedings of the ACL-02 workshop on Word sense disambiguation: recent successes and future directions*. 74–80.

[24] Jing Li, Zhenchang Xing, Deheng Ye, and Xuejiao Zhao. 2016. From discussion to wisdom: web resource recommendation for hyperlinks in stack overflow. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 1127–1133.

[25] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*. 83–94.

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[27] Kien Luong, Ferdian Thung, and David Lo. 2021. Disambiguating Mentions of API Methods in Stack Overflow via Type Scoping. In *ICSME*. IEEE.

[28] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 260–270. https://doi.org/10.1109/ASE.2015.42

[29] Alexios Mandalios, Konstantinos Tzamaloukas, Alexandros Chortaras, and Giorgos Stamou. 2018. Geek: Incremental graph-based entity disambiguation. In *LDOW@ WWW*.

[30] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[31] Andrian Marcus and Jonathan I Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*. IEEE.

[32] Sarah Meldrum, Sherlock A Licorish, and Bastin Tony Roy Savarimuthu. 2017. Crowdsourced knowledge on stack overflow: A systematic mapping study. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 180–185.

[33] Anh Tuan Nguyen, Peter C Rigby, Thanh Nguyen, Dharani Palani, Mark Karanfil, and Tien N Nguyen. 2018. Statistical translation of English texts to API code templates. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 194–205.

[34] Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2018. Complementing Global and Local Contexts in Representing API Descriptions to Improve API Retrieval Tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 551–562. https://doi.org/10.1145/3236024.3236036

[35] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Institute of Technology, Tech. Rep* 11 (2012).

[36] Siddharth Patwardhan, Satanjeev Banerjee, and Ted Pedersen. 2005. Senserelate:: Targetword-A generalized framework for word sense disambiguation. In *ACL*, Vol. 2005. 73–76.

[37] Hung Phan, Hoan Anh Nguyen, Ngoc M Tran, Linh H Truong, Anh Tuan Nguyen, and Tien N Nguyen. 2018. Statistical learning of api fully qualified names in code snippets of online forums. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 632–642.

[38] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the content of github readme files. *Empirical Software Engineering* 24, 3 (2019), 1296–1327.

[39] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *SANER 2016*, Vol. 1. IEEE.

[40] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34. https://doi.org/10.1109/MS.2009.193

[41] Adriano M Rocha and Marcelo A Maia. 2016. Automated API documentation with tutorials generated from Stack Overflow. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*. 33–42.

[42] Michał Rój. 2011. Exploiting User Knowledge during Retrieval of Semantically Annotated API Operations. In *Proceedings of the Fourth Workshop on Exploiting Semantic Annotations in Information Retrieval* (Glasgow, Scotland, UK) *(ESAIR '11)*. Association for Computing Machinery, New York, NY, USA, 21–22. https://doi.org/10.1145/2064713.2064726

[43] CM Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K Roy. 2019. Learning from examples to find fully qualified names of api elements in code snippets. In *ASE*. IEEE.

[44] Ramin Shahbazi, Rishab Sharma, and Fatemeh H Fard. 2021. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. *arXiv preprint arXiv:2103.10668* (2021).

[45] Megan Squire. 2015. " Should We Move to Stack Overflow?" Measuring the Utility of Social Media for Developer Support. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 219–228.

[46] Thomas Steiner, Ruben Verborgh, Joaquim Gabarró Vallés, and Rik Van de Walle. 2013. Adding meaning to social network microposts via multiple named entity disambiguation apis and tracking their data provenance. *International Journal of Computer Information Systems and Industrial Management* 5 (2013), 69–78.

[47] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. 643–652.

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

[49] Gias Uddin and Foutse Khomh. 2019. Automatic mining of opinions expressed about apis in stack overflow. *IEEE Transactions on Software Engineering* (2019).

[50] Gias Uddin and Martin P Robillard. 2015. How API documentation fails. *Ieee software* 32, 4 (2015), 68–75.

[51] Liliane do Nascimento Vale and Marcelo de Almeida Maia. 2021. Towards a question answering assistant for software development using a transformer-based language model. *arXiv preprint arXiv:2103.09423* (2021).

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[53] Pradeep K Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed E Hassan. 2016. What do client developers concern when using web apis? an empirical study on developer forums and stack overflow. In *2016 IEEE International Conference on Web Services (ICWS)*. IEEE, 131–138.

[54] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. Transˆ 3: A transformer-based framework for unifying code summarization and code search. *arXiv preprint arXiv:2003.03238* (2020).

[55] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[56] Deheng Ye, Lingfeng Bao, Zhenchang Xing, and Shang-Wei Lin. 2018. APIReal: an API recognition and linking approach for online developer forums. *Empirical Software Engineering* 23, 6 (2018), 3129–3160.

[57] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/2884781.2884862

[58] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.

[59] Stefan Zwicklbauer, Christin Seifert, and Michael Granitzer. 2013. Do We Need Entity-Centric Knowledge Bases for Entity Disambiguation?. In *Proceedings of the 13th International Conference on Knowledge Management and Knowledge Technologies*. 1–8.