# CLEAR: Contrastive Learning for API Recommendation

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

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

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

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

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

## ABSTRACT

Automatic API recommendation has been studied for years. There are two orthogonal lines of approaches for this task, i.e., information-retrieval-based (IR-based) and neural-based methods. Although these approaches were reported having remarkable performance, our observation shows that existing approaches can fail due to the following two reasons: 1) most IR-based approaches treat task queries as bags-of-words and use word embedding to represent queries, which cannot capture the sequential semantic information. 2) both the IR-based and the neural-based approaches are weak at distinguishing the semantic difference among lexically similar queries.

In this paper, we propose CLEAR, which leverages BERT sentence embedding and contrastive learning to tackle the above two issues. Specifically, CLEAR embeds the whole sentence of queries and Stack Overflow (SO) posts with a BERT-based model rather than the bag-of-word-based word embedding model, which can preserve the semantic-related sequential information. In addition, CLEAR uses contrastive learning to train the BERT-based embedding model for learning precise semantic representation of programming terminologies regardless of their lexical information. CLEAR also builds a BERT-based re-ranking model to optimize its recommendation results. Given a query, CLEAR first selects a set of candidate SO posts via the BERT sentence embedding-based similarity to reduce search space. CLEAR further leverages a BERT-based re-ranking model to rank candidate SO posts and recommends the APIs from the ranked top SO posts for the query.

Our experiment results on three different test datasets confirm the effectiveness of CLEAR for both method-level and class-level API recommendation. Compared to the state-of-the-art API recommendation approaches, CLEAR improves the MAP by 25%-187% at method-level and 10%-100% at class-level.


*ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510159

## CCS CONCEPTS

• **Computing methodologies** → **Semantic networks**; • **Applied computing** → **Document searching**; • **Information systems** → **Recommender systems**.

## KEYWORDS

API recommendation, contrastive learning, semantic difference

**ACM Reference Format:**
Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: Contrastive Learning for API Recommendation. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510159

## 1 INTRODUCTION

Over the past decades, open-source software development has received extensive attention from the software engineering community. This attention leads to a tremendous demand for already devised libraries or APIs which facilitate software development and maintenance. Developers often search for existing APIs or code snippets on the Internet to obtain the functions they wish to implement [51].

To help with API search, many automated API recommendation approaches have been proposed [11, 14, 16, 22, 24, 34, 35, 39]. There are two orthogonal lines of approaches for this task, i.e., information retrieval based, e.g., BIKER [14], and neural-based methods, e.g., DeepAPI [11]. BIKER [14] uses bag-of-word-based word embedding (i.e., a word2vec model built on Java SO posts) and IDF (inverse document frequency) vocabulary to calculate the similarity score between two text descriptions and then leverages a query's similarity with both SO posts and API documentations to recommend appropriate APIs for the query. DeepAPI [11] formulates the API recommendation as a machine translation problem, i.e., given a natural language query, it aims to translate it into an API sequence. Specifically, it adapts a Recurrent Neural Network (RNN) Encoder-Decoder model to encode a query into a fixed-length context vector and recommends an API sequence based on the context vector for the query. Although these approaches achieved remarkable performance, by replicating these studies, we found two major problems that can affect their effectiveness.

The first problem is that these IR-based approaches (e.g., BIKER) treat queries and SO posts as bag-of-words and use word embedding to represent queries [12], which cannot capture the semantic-related sequential information. For example, given a real-world query "*Convert String to Calendar Object in Java*"[1], BIKER cannot recommend the correct API and the top API recommended by BIKER is "`java.time.LocalDate.parse`" from the most similar post identified by BIKER, i.e., "*Convert Java Gregorian Calendar to String*"[2], whose intent is opposite to the intent of the query. BIKER fails to retrieve the correct answer for the above query because of its bag-of-words-based representation, which cannot capture the semantic-related sequential information. To properly represent the semantic-related sequential information of the text descriptions, the embedding of queries and SO posts has to be considered comprehensively instead of using bag-of-words.

The second problem is that both the IR-based and the neural-based approaches are weak at distinguishing the semantic difference among quires that are lexically similar. For example, given a real-world query "*Filereader.read() method not working*"[3], neither BIKER nor DeepAPI can recommend a correct API. Specifically, The most likely API recommended by BIKER is "`java.io.RandomAccessFile.read`" from the post "*BufferedReader read() not working*"[4], as the text descriptions of the query and the post are almost identical except the terminology "`Filereader`" and "`BufferedReader`". However, the answer to this query is "`java.io.OutputStreamWriter.flush`". The root cause of such a failure of BIKER is that the two quires are lexically close but semantically different. BIKER's word2vec embedding relies on the context of the words in a text description. However, the above example shows that only using the context of the words is not enough to distinguish the semantic of the query in API recommendation tasks. For DeepAPI, we experiment with the above two queries "*Filereader.read() method not working*" and "*BufferedReader read() not working*", while DeepAPI generates the same API sequence for both queries, i.e., {*"String.length", "Object.toString"*}, which is incorrect as these two queries have different semantics. One of the reasons for such a failure is that DeepAPI uses an RNN Encoder-Decoder base architecture to encode every query into a fixed-length context vector and generates an API sequence based on the overall context of the query. Thus, due to the above nature of RNN, DeepAPI often fails for similar queries that have different key words [1, 10].

To alleviate the above two problems, we propose CLEAR, an API recommendation approach based on BERT sentence embedding [7] and contrastive learning [25]. Specifically, to solve the first issue, CLEAR uses a BERT-based model to embed text descriptions of queries and SO posts, which produces the embedding of the whole sentence of an API query while taking sequential information into consideration rather than combining the embedding of each word (i.e., bag-of-words). For solving the second issue, CLEAR uses contrastive learning to train the BERT sentence embedding model for learning semantically equivalent representation of queries or SO posts regardless their lexical information. Given a query, CLEAR first selects a set of candidate SO posts via the BERT sentence embedding-based similarity to reduce search space. CLEAR further leverages a BERT-based classification model to re-rank candidate SO posts and recommend the APIs from the ranked top SO posts for the query.

In order to evaluate the effectiveness of CLEAR, we re-use the dataset from BIKER [14]. Specifically, we have developed three test sets derived from BIKER's dataset for testing. The first is BIKER's manually created test dataset, the second is randomly selected 1k sample SO posts to alleviate potential human bias. Since around 10% posts in SO contain multiple APIs, in order to test the performance on the scenario of multi-API answers, we added a third test dataset, which is 1K randomly selected sample SO posts with multiple APIs in answers. We use the corpus that excludes these testing data as our training dataset to train CLEAR. The results show that CLEAR outperforms the state-of-the-art information retrieval based and neural-based approaches (i.e., BIKER [14] and DeepAPI [11] respectively) significantly at both method- and class-level on all three test sets. We also conduct a case study to evaluate CLEAR against the latest SO posts, and the results confirm the effectiveness and practical values of CLEAR. This paper makes the following contributions:

- We propose CLEAR, a novel API recommendation approach, which uses the BERT sentence embedding model to represent queries for capturing sequential semantic information and leverages contrastive training to train the BERT model for learning precise semantic representation of queries regardless of their lexical information.
- We evaluate CLEAR using three different test datasets, including test data from previous studies, 1k randomly selected SO posts, and 1k randomly selected SO posts with multi-API answers. Our experiment results confirm that CLEAR can significantly outperform the state-of-the-art baselines.
- We conduct a case study on the latest SO posts to evaluate the performance of CLEAR and our results suggest the practical value of CLEAR.
- We release the source code of CLEAR and the dataset of our experiments to help other researchers replicate and extend our study[5].

The rest of this paper is structured as follows. Section 2 describes the background of this study. Section 3 presents the framework of the proposed CLEAR. Section 4 introduces experimental design, baselines, and research questions. Section 5 analyzes the experiment results. Section 6 discusses open questions and the threats to the validity of this work. Section 7 surveys the related work and Section 8 summarizes this paper.

## 2 BACKGROUND

### 2.1 Language Embedding

Language embedding technique is a method for converting words or sentences into numerical vectors [7, 28, 36]. The deep learning-based language models have been widely examined to be useful in capturing implicit semantics for natural language sentences.

[1] https://stackoverflow.com/questions/5301226/convert-string-to-calendar-object-in-java

[2] https://stackoverflow.com/questions/24741696/convert-java-gregorian-calendar-to-string

[3] https://stackoverflow.com/questions/36427839/filereader-read-method-not-working

[4] https://stackoverflow.com/questions/43190995/bufferedreader-read-not-working

[5] Reproduction package link: https://github.com/Moshiii/CLEAR-replication

There exist studies of language embedding on both word-level [20, 27] and sentence-level [7, 28, 36]. Typical deep learning language embedding models include GPT [32] and BERT [7].

GPT [32] introduces minimal task-specific parameters and is trained on the downstream tasks by simply fine-tuning all pre-trained parameters. BERT [7] is deep learning language embedding based on transformer units. It uses a 12-layer or 24-layer transformer layer with a multi-head attention mechanism as feature extraction, and then uses a regression function to generate the final output. BERT model can be used for multiple tasks, e.g., sentence embedding, classification, question-answer tasks, sentence tagging, etc., with different minor adaptions [7].

In this paper, we use the BERT model for two different tasks. First, we use BERT as sentence embedding to represent the text of queries and SO posts for preserving their semantic information regardless of their lexical information. Second, we use BERT as a binary SO post classifier to re-rank the retrieved SO posts for a given query. For both the two tasks, we use RoBERTa model, a state-of-the-art BERT variant [17]. For sentence embedding, we adopt the contrastive learning process to train the model, we provide an input sample to the model and take the output vector of the model as the sentence embedding of the input. For re-ranking posts, following existing work [7], we use the joint embedding training process to train the classifier, which takes paired posts as input and the label is whether or not they have the same APIs.

The difference of RoBERTa to the original BERT model is that RoBERTa applied different training processes and distillations in training [29], which reduces the number of parameters while increasing the robustness of the BERT model.

## 2.2 Contrastive Learning

Contrastive learning [25] is a deep neural network training process that takes paired sentences as input and uses the similarity in the paired sentences as labels. The training goal is to learn the relationship between sentences, i.e., whether two sentences are semantically similar regardless of their lexical similarity. Hoffer et al. [13] proposed the triplet network for contrastive training. It requires a triplet $(S, P, N)$ as the input, where $S$ corresponds to the original query, $P$ refers to the positive equivalent of $S$, and $N$ is the negative one.

In this work, we use contrastive learning to train a RoBERTa [17] model for sentence embedding. For a given post in the training data, its positive posts are posts with the same answer and negative posts are posts with different answers.

## 2.3 Joint Embedding Training

Joint embedding training [7] was widely used to train BERT as a classification model. Figure 1 shows the architecture of joint embedding training for BERT. BERT [7] provides a special token $[SEP]$, which allows two posts to be concatenated as input. In joint embedding training, $[SEP]$ is used to identify the end of the first post. The process of joint embedding training is fine-tuning the model with pairs of posts to the target that if given two semantic equivalent posts, the model returns 1, otherwise returns 0. The loss function we use for joint embedding training is the classic

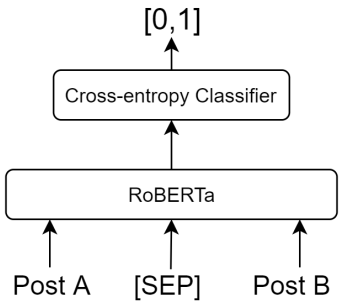


**Figure 1: Joint embedding training**

cross-entropy loss function (i.e., *Loss*):

$$Loss = -(y * log(\hat{y}) + (1 - y) * log(1 - \hat{y})) \quad (1)$$

where $y$ indicates whether the given two posts are semantically equivalent, and $\hat{y}$ is the prediction of the re-ranking model.

In this work, we leverage joint embedding training to train a RoBERTa based classification model to re-rank the retrieved SO posts for a given query.

# 3 APPROACH

Figure 2 shows the pipeline of CLEAR, which consists of two parts: language model building (section 3.1) and searching relevant APIs (section 3.2). The language model building process contains four steps, i.e., post triplets construction (Section 3.1.1), BERT sentence embedding with contrastive learning (Section 3.1.2), candidate posts filtering (Section 3.1.3), and the joint embedding training based re-ranking model (Section 3.1.4).

## 3.1 Building BERT-base Language Models

*3.1.1* ***Post Triplets Construction.*** The format of the training data used in the contrastive training process is different from the traditional natural language processing tasks, e.g., sentiment analysis, where the inputs are sentences and the outputs are the labels. Contrastive training requires triplets as inputs [13]. Every single triplet is a combination of three posts, which are an input query $S$, a positive sample post $P$ that is semantically equivalent to $S$, and a negative sample post $N$ that is not related to $S$ and $P$. Therefore, the training corpus needs to be converted to triplets. For example, given an input query "*Java string split with multiple delimeters*", the triplet $(S, P, N)$ can be ("*Java string split with multiple delimeters*","*How to split a path using StringTokenizer?*","*How to load a file across the network and handle it as a String*"). Algorithm 1 shows the process of generating training triplets.

Our triplets generation algorithm has two parameters, i.e., $p$ is the number of positive samples and $n$ is the number of negative sampling for a training instance. When generating the triplets, each question needs to be paired with positive and negative samples. For each question $item$ in $T$, we use function *get_equivalent_subset()* to get its positive posts, i.e., posts that have the same answer with $item$. In addition, we consider posts that have a different answer from $item$ as the negative posts of $item$.

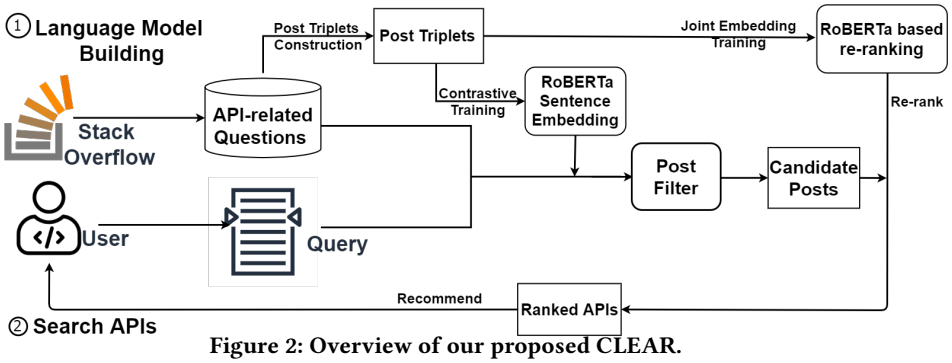


**Figure 2: Overview of our proposed CLEAR.**

**Algorithm 1:** Triplets Generator

```
Result: Tuple list of element (S, P, N)
def getTriplets(p: int, n: int, T: list<question, answer>):
    result_list = new list() // initialize empty result list
    for item in T do
        S = item[0] //question sentence
        answer = item[1]
        T_P= get_equivalent_subset(T, answer)
        T_N = set(T) - set(T_P)
        P_list = random_sample(T_P, p)
        N_list = random_sample(T_N, n)
        for item P in P_list do
            for item N in N_list do
                result_list.append(S, P, N)
            end
        end
    end
    return result_list
```

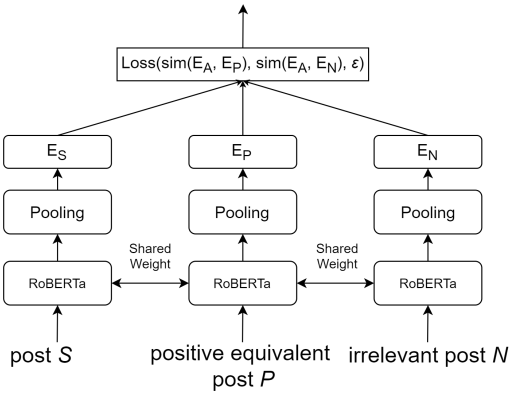


**Figure 3: Architecture for contrastively trainning RoBERTa based sentence embedding model**

Note that, the ratio between positive and negative samples is important in contrastive training, different configurations may impact the result significantly [15, 41]. In our algorithm, To find the best configurations, i.e., $p$ and $n$, we perform a grid search with a list of candidate values for both $p$ and $n$, which are 1, 3, 5, 10, and 15. We use $p$ and $n$ that can achieve the best performance in our experiment (details are in Section 4.2). For APIs that do not have $p$ positive samples, we use all their positive samples. We perform random sampling on the APIs that exceed $p$ or $n$ to limit the number of positive or negative samples.

*3.1.2* ***BERT Sentence Embedding Model.*** In this step, we use contrastive training to train the RoBERTa based sentence embedding model with the post triplets created in Section 3.1.1. The goal of this process is to learn a semantic presentation, with which similar samples stay close to each other, while dissimilar ones are far apart. Figure 4 shows an illustration for this process. In the Figure, green points are positive posts that have the same API "`Arrays.asList`" with query $S$, and the red points are negative posts of $S$. With contrastive learning, the center green point $S$ plays the role of an anchor, the positive samples are pulled towards the anchor and the negative samples are pushed away from the anchor.

Figure 3 shows the architecture of the contrastive learning used in our work, in which the RoBERTa [17] model is the base model for sentence embedding, and we use a Pooling layer to connect the RoBERTa model and the triple network. Triple network has two layers, the first layer is three identical deep neural network models for feature extraction of input sentences. The feature extraction layer can also be replaced with other models or algorithms. The second layer of the triplet network is a loss function based on the cosine distance operator. The purpose of the loss function is to minimize the distance between similar sentences and maximize the distance between unrelated sentences. The training objective is to fine-tune the network so that the distance between the question $S$ and the positive question $P$ is closer than the distance between the question $S$ and the negative question $N$. Formally, the training objective is to minimize the following function:

$$max(||E_s - E_p|| - ||E_s - E_n|| + \epsilon, 0) \quad (2)$$

where $E_s$, $E_p$, and $E_n$ are the sentence embeddings of question $S$, $P$, and $N$. $\epsilon$ is the margin of the distance between $S$ and $N$. By default, $\epsilon$ is set to 1, which means that the cosine distance between a question and its irrelevant question should be 1.

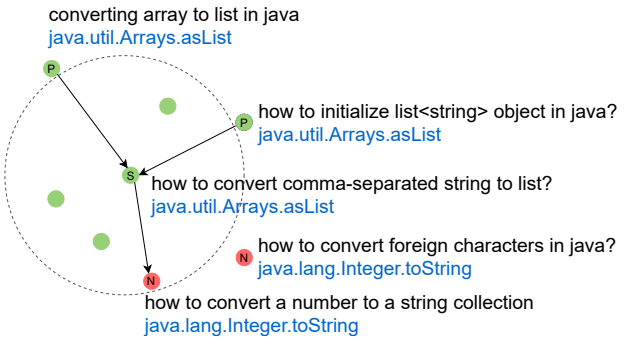


**Figure 4: Contrastive training for a single post.**

*3.1.3* ***Candidate Posts Filter.*** In this step, with the BERT sentence embedding built-in Section 3.1.2, we further filter out irrelevant questions for a given query and keep top-k questions for detecting candidate APIs. Following existing work [14], CLEAR keeps top-50 similar questions as the candidates, since retrieving too many questions can introduce noise to the recommendation process. In this step, we use the euclidean distance between two questions as the metric to filter out irrelevant questions.

Note that, although our experimental results show that directly using the 50 candidate questions from the filter for API recommendation can achieve better performance than both BIKER and DeepAPI (details are in Section 5.1), we observe that there exist noisy questions in the retrieved 50 candidate questions from the filter (one of the possible reasons is the low quality of SO posts [30]), which could hurt the performance of API recommendation. Thus, a re-ranking model for the candidate posts is needed and details are in the next section.

*3.1.4* ***Candidate Post Re-ranking Model.*** The objective of our filter model (details are in Section 3.1.3) is to filter out the number of irrelevant posts from the entire search space, while this re-ranking model is to optimize the ranking of the left $k$ candidate posts from the filter model.

For semantic embedding re-ranking tasks, we choose the same BERT model, i.e., RoBERTa [17], the state-of-the-art BERT-based model for semantic embedding re-ranking tasks, as the base model. Then, we fine-tune it with joint embedding training, which turns the RoBERTa into a classification model (details are in Section 2), the label is whether two posts have the same APIs. For training the model, we first use the filter model (details are in Section 3.1.3) to find the top-50 similar posts, i.e., $T_s = \{s_1, s_2, s_3, ...s_{50}\}$, for a post $p$ in our training dataset. We then create 50 pairs from the post, i.e., $pairs = \{< p, s_1 >, < p, s_2 >, < p, s_3 >, ... < p, s_{50} >\}$ and the label of each pair is whether they have the same APIs. In total, we have around 1.7M pairs to train the RoBERTa based classification model. We use the predicted possibility to rank the 50 candidate posts.

## 3.2 Search APIs

Given a natural language described query $Q$, the first step is to retrieve the top-k candidate questions from SO. CLEAR first uses the trained RoBERTa based sentence embedding model to transform it into an embedding. CLEAR then uses the filter model to filter out irrelevant posts and get a list of candidates' posts based on the BERT sentence embedding. Then, in the re-ranking phase, the re-ranking model calculates the probability that $Q$ and a given candidate post have the same label, we use the probability to rank the 50 candidate posts. We then extract the APIs from the ranked posts and output them as the recommendation to the query $Q$. After obtaining the ranked list of candidate APIs, CLEAR also summarizes supplementary information for $Q$ to describe the API usage examples and help users decide which API should be chosen for their tasks. The supplementary information summarized by CLEAR considers two aspects, i.e., the title of similar questions and code snippets from these questions.

Note that, CLEAR recommends APIs at method-level by default. It can be easily adapted to class-level recommendations as well. In the case of API class searching, we remove the method name of the candidate API to adjust the candidate API to the class level.

# 4 EXPERIMENT DESIGN

## 4.1 Dataset

To evaluate the performance of CLEAR, we reuse SO data from the state-of-the-art approach BIKER [14], which were collected from the official data dump of SO by following criteria: 1) the question is related to Java JDK programming, 2) the question should have a positive score, and 3) at least 1 answer to the question contains API entities and the answer's score should be positive.

The APIs were extracted from the code snippets in markdown scripts of the accepted answers in SO. In a markdown script, code snippets are wrapped by <code> tags. One can use regular expressions to localize the code snippets and further extract the APIs. In total, BIKER's dataset contains 33K Java-related questions. BIKER also provided a test dataset for evaluating its performance, which was manually created with a set of well-designed criteria, e.g., one of their criteria is the score of the question itself should be at least five, the details about the process are in their Section 4. The test data contains 413 questions along with their ground truth APIs. We use the title of these 413 questions as the query for API search.

Note that, BIKER's test dataset mainly contains SO posts with high quality, which cannot reflect the overall quality of SO posts. Thus, we have also created two different random test datasets which contain randomly selected SO posts for removing human bias (details are in Section 4.3).

## 4.2 Experiment Settings

We use Google Colab [2] professional version for fine-tuning the models. The CPU we use is two Intel Xeon 2.20GHz CPU with 5G cache. The GPU resource we use is one NVIDIA V100 graphic card with 13G memory. We fine-tune the filtering model and the re-ranking model for five epochs each and then select the model with the best performance on the validation set.

The triplets generation algorithm in CLEAR has two parameters, i.e., the number of positive samples ($p$) and the number of negative samples ($n$), which could affect the performance of CLEAR. To find the best values of these two parameters, we tune them together. For $p$ and $n$, we experiment with five discrete values, i.e., 1, 3, 5, 10, and 15, which results in a combination of 25 model

**Table 1: Performance comparison of different (P)ositive sampling and (N)egative sampling settings.**

| P\N | 1 | 3 | 5 | **10** | 15 |
|---|---|---|---|---|---|
| 1 | 0.004 | 0.032 | 0.027 | 0.036 | 0.027 |
| 3 | 0.184 | 0.332 | 0.392 | 0.463 | 0.416 |
| 5 | 0.376 | 0.512 | 0.552 | 0.766 | 0.76 |
| **10** | 0.524 | 0.704 | 0.652 | **0.828** | 0.784 |
| 15 | 0.624 | 0.684 | 0.736 | 0.72 | 0.784 |

configurations. Because the fine-tuning on the full data is very time-consuming, we perform the grid search on the model with a quarter of the training data. We train the filtering model until full convergence or up to 5 epochs to sufficiently train the models. The random seed is locked across the models to make sure the random sampling on positive and negative samples is consistent. We randomly select 5K posts as the test data for tuning these two parameters, and we use the accuracy of our filter model as a metric during our tuning. Following existing studies [6, 40], we use the *accuracy*@1 as the metric for parameter tuning, which is calculated as $\#(first\ match\ is\ correct)/\#(test\ instances)$.

Table 1 shows the result of *accuracy*@1 based on different parameter settings, in which the row and column indices are the numbers of positive and negative samples respectively. Overall, the performance of CLEAR increases with the increase of positive and negative samples, and the performance of CLEAR reaches the peak at the point where the number of positive and negative samples are both equal to 10. Thus, we set 10 positive samples and 10 negative samples for each training instance when training CLEAR in our experiments. In the case that there are less than 10 positive and negative samples, we include all positive and negative samples.

## 4.3 Evaluation Datasets

To comprehensively evaluate the performance of CLEAR, we adopt three test datasets covering three different scenarios, i.e., high-quality SO posts (i.e., BIKER's test dataset), real-wold random SO posts, and SO posts with multi-API answers as our observation shows that around 10% posts in SO contain multiple APIs. The details of the three datasets are as follows:

- **BIKER test dataset:** is the evaluation dataset of BIKER, which contains 413 manually selected and verified SO queries with API answers.
- **Random test dataset:** contains 1K random selected SO queries with API answers from BIKER's training dataset.
- **Multi-API test dataset:** contains 1K random SO queries with multi-API answers from BIKER's training dataset.

During our experiments, questions from the test datasets and their duplicate questions were excluded from the training dataset.

## 4.4 Baselines

We compared CLEAR with BIKER [14], RACK [35], and DeepAPI [11], which are three state-of-the-art API recommendation techniques. To show the impact of contrastive training, we also introduce a variant of the filter model without adopting the contrastive training, which is the pre-trained RoBERTa model. Note that, BIKER and our CLEAR share a common procedure, i.e., a filter model to retrieve top k candidate posts and a re-ranking model to re-rank the candidate posts. Thus, we also introduce the filer models of BIKER and our CLEAR as the baselines.

**Baseline1 (BIKER)** [14]: first uses a mixture of TF-IDF and a trained Word2vec model to calculate the similarity of a given query and the SO posts and then the top 50 posts are selected as the candidates. Finally, it re-ranks the 50 candidates by using the similarity between the query and the corresponding official API document descriptions. To comprehensively compare it with CLEAR, we employ two related baselines, i.e., BIKER-filter (BIKER without re-ranking) and BIKER-complete (the whole approach).

**Baseline2 (RACK)** [35]: is a keyword-API mapping system that recommends APIs by matching keywords from the query. The keyword-API is constructed by mining the statistical relationship between the SO questions and the accepted answers of questions. Please note that RACK only recommends API at the class level.

**Baseline3 (DeepAPI)** [11]: models API recommendation task as a machine translation problem. It uses a Recurrent Neural Network (RNN) Encoder-Decoder model to encode a given query into a fixed-length context vector, and generate an API-method sequence based on the context vector. The author of DeepAPI provided an online tool for testing and evaluation. However, the website is not available currently due to the budget limit. Initially, we contacted the authors for their trained models, unfortunately, the author claimed that they did not maintain the trained models anymore. Then, we used its reproduction package[6] and rigorously follow its instruction to re-train the DeepAPI model from scratch with its dataset. The training process takes 15 days and we achieve similar performance (regarding BLUE scores) as reported in the paper of DeepAPI. The reproduction model represents the best effort we made to reproduce the DeepAPI model. In this work, the evaluation of the DeepAPI model is performed on the reproduced model.

**Baseline5 (Pre-trained RoBERTa filter)**: is the pre-trained RoBERTa model. We compare CLEAR-filter with RoBERTa to explore the performance increase introduced by contrastive learning. We use the same pre-trained RoBERTa model as used in CLEAR.

**Baseline5 (CLEAR-filter)**: Since CLEAR has two steps, i.e., the filter model, and the re-ranking model, we separate the filter model from the re-ranking model to show the performance increase introduced by both of them.

## 4.5 Performance Measures

Following existing studies [14], we use Mean reciprocal rank(MRR) [31, 45], Mean average precision(MAP) [38], *Precision*@$k$, and *Recall*@$k$, to evaluate the performance of API recommendation approaches. MRR and MAP are the widely accepted measurements for information retrieval. MRR measures the effort needed to find the first correct answer in the recommended list and MAP considers the ranks of all correct answers.

We also evaluate the performance with *Precision*@$k$ and *Recall*@$k$, where $k$ can be 1, 3, 5, and 10. For the search result of a query, precision and recall can be defined as follows:

$$Precision@k = \frac{\#(relevant\ items\ retrieved)@k}{\#(retrieved\ items)} \quad (3)$$

[6] https://github.com/guxd/deepAPI

$$Recall@k = \frac{\#(relevant\ items\ retrieved)@k}{\#(relevant\ items)} \quad (4)$$

where the $\#(relevant\ items\ retrieved)$ refers to the number of correctly recommended API, the $\#(retrieved\ items)$ refers to the number of total retrieved APIs, and the $\#(relevant\ items)$ refers to the number of APIs in the answers of the queries.

## 4.6 Research Questions

To evaluate the performance of CLEAR, we design experiments to answer the following research questions:

**RQ1:** *How effective is CLEAR comparing with existing API recommendation baselines at method-level?*

**RQ2:** *How effective is CLEAR comparing with existing API recommendation baselines at class-level?*

**RQ3:** *How does random sampling of triplet generation affect the performance of CLEAR?*

In RQ1 and RQ2, we set out to investigate the performance of the CLEAR on method- and class-level API recommendation tasks. To demonstrate its advantages, we compare CLEAR with state-of-the-art baselines (details are in Section 4.4). In RQ3, we explore the impact of the random sampling process in the triplet generation algorithm (details are in Section 3.1.1) on the performance of CLEAR.

## 4.7 Statistical Testing

In this paper, we use a parametric test to check the statistically significant difference in performance of different API recommendation baselines. We use the parametric Wilcoxon signed ranked test [50], which has been widely used in many software engineering studies [21, 46–48]. The advantage of the Wilcoxon test is that it does not require the results to follow any specific distribution. A p-value smaller than 0.05 indicates that the difference between the two baselines' performance is statistically significant.

# 5 RESULT ANALYSIS

This section presents our experiment results and answers the three research questions asked in Section 4.6 regarding the effectiveness of CLEAR at method-level API recommendation (Section 5.1) and class-level API recommendation (Section 5.2) and the impact of randomness in CLEAR (Section 5.3).

## 5.1 RQ1: Effectiveness of CLEAR at Method-level

**Experimental Method**. To answer this research question, we compare CLEAR with the baselines listed in Section 4.4 on the three different test datasets listed in Section 4.3. Note that, we exclude RACK in this research question as it recommends API at class-level only. Since BIKER's authors have published the replication package[7], we directly use it to conduct experiments and compare with CLEAR. For DeepAPI, as we described in Section 4.4, we use the re-trained model for our experiments. Since DeepAPI recommends API sequence for a given query, we consider a recommendation is correct if any one of the APIs in the sequence is the ground truth API of the query (the same comparison manner has also been used in the comparison of BIKER and DeepAPI in BIKER's paper [14]).

**Results**. Table 2 shows the result of CLEAR compared with the other baselines. As shown in the Table 2, overall CLEAR outperforms both BIKER (including both its filter model and re-ranking model) and DeepAPI. Note that, BIKER has the same performance reported in this work and its original paper [14]. However, different from the comparison reported in BIKER's paper [14], where DeepAPI's MRR and MAP are 0.183 and 0.155, in this study DeepAPI reports much worse performance, i.e., all MRRs and MAPs are below 0.1. The reason is that in paper [14], DeepAPI was evaluated on the online tool released by DeepAPI's authors, we re-trained DeepAPI with its reproduction package (details are in Section 4.4). On BIKER test data, the $recall@1$ of CLEAR-complete is 0.6309, indicating that there is at least one right answer in the first candidates in 63.09% cases. Comparing the BIKER-filter model and CLEAR-filter model, the CLEAR-filter model outperforms the BIKER-filter model by 46.43% and 50.18% on MAR and MAP. In terms of precision and recall, CLEAR-filter model improves the $precision@1, 3, 5, 10$ by 51.45%, 120.44%, 151.37%, 166.49%, $recall@1, 3, 5, 10$ by 52.61%, 31.63%, 22.25%, 4.005% respectively, which shows the effectiveness of our filter model.

On the random test data, CLEAR-complete model outperforms BIKER-complete model in all the measurements. Comparing to BIKER-complete model, CLEAR-complete model improves by 185.88% on MRR, 195.15% on MAP, 314.94%, 541.88% 732.24% 1132.05% on $precision@1, 3, 5, 10$, and 326.29%, 180.50%, 133.18%, 87.45% on $recall@1, 3, 5, 10$ respectively. On multi-API test data, CLEAR-complete mode outperforms BIKER-complete mode by 104.09% on MRR and 105.24% on MAP. In terms of precision and recall, CLEAR-complete improves the $precision@1, 3, 5, 10$ by 287.09%, 506.31%, 711.52%, 1126.38% and $recall@1, 3, 5, 10$ by 301.99%, 165.70%, 130.77%, 87.35% respectively. Compared to the RoBERTa model, CLEAR's filter model achieves better performance on all the three test datasets, which indicates that contrastive learning can help learn a precise semantic representation of programming tasks.

We have also conducted the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of CLEAR and baselines. the test result suggests that CLEAR achieves significantly better performance than all the baselines.

> CLEAR significantly outperforms the state-of-the-art baselines at method-level API recommendation and CLEAR's performance remains stable across different test datasets.

## 5.2 RQ2: Effectiveness of CLEAR at Class-level

**Experimental Method**. To answer this research question, we perform the same evaluation method on the baselines and CLEAR. We use the same three test datasets with API methods removed to compare API answers at the class level. To compare with RACK, we run experiments with RACK's replication[8]. For both BIKE and DeepAPI, we use the same manner as the experiment at method-level API recommendation in Section 5.1.

[7] https://www.dropbox.com/s/fr4gdbyfn58ytm8/BIKER.zip?dl=0

[8] https://github.com/masud-technope/RACK-Replication-Package

**Table 2: Performance comparison at method-level (RQ1)**

| Method-level | | | BIKER-filter | BIKER-complete | DeepAPI | RoBERTa | CLEAR-filter | CLEAR-complete |
|---|---|---|---|---|---|---|---|---|
| **BIKER test data** | MRR | | 0.4318 | 0.6225 | 0.0313 | 0.4098 | 0.6319 | **0.7551** |
| | MAP | | 0.4260 | 0.6175 | 0.0102 | 0.4088 | 0.6398 | **0.7655** |
| | Precision | P@1 | 0.2777 | 0.4642 | 0.0088 | 0.2341 | 0.4206 | **0.4682** |
| | | P@3 | 0.2328 | 0.2486 | 0.0073 | 0.2632 | 0.5132 | **0.5502** |
| | | P@5 | 0.2071 | 0.1698 | 0.0140 | 0.2563 | 0.5206 | **0.5531** |
| | | P@10 | 0.1928 | 0.0956 | 0.0123 | 0.2305 | 0.5138 | **0.5563** |
| | Recall | R@1 | 0.2678 | 0.4503 | 0.0029 | 0.2321 | 0.4087 | **0.6309** |
| | | R@3 | 0.5019 | 0.7142 | 0.0066 | 0.4980 | 0.6607 | **0.7638** |
| | | R@5 | 0.5972 | **0.8134** | 0.0227 | 0.6130 | 0.7301 | 0.7956 |
| | | R@10 | 0.7440 | **0.9166** | 0.0403 | 0.7182 | 0.7738 | 0.8551 |
| **Random test data** | MRR | | 0.2448 | 0.2813 | 0.0336 | 0.2912 | 0.7573 | **0.8042** |
| | MAP | | 0.2357 | 0.2724 | 0.0104 | 0.2855 | 0.7612 | **0.8040** |
| | Precision | P@1 | 0.1420 | 0.1740 | 0.0080 | 0.1940 | 0.6680 | **0.7220** |
| | | P@3 | 0.1266 | 0.1103 | 0.0057 | 0.1746 | 0.6669 | **0.7080** |
| | | P@5 | 0.1160 | 0.0830 | 0.0131 | 0.1673 | 0.6495 | **0.6909** |
| | | P@10 | 0.1074 | 0.0546 | 0.0137 | 0.1524 | 0.6233 | **0.6727** |
| | Recall | R@1 | 0.1298 | 0.1620 | 0.0023 | 0.1783 | 0.6383 | **0.6906** |
| | | R@3 | 0.2673 | 0.3011 | 0.0052 | 0.3093 | 0.8078 | **0.8446** |
| | | R@5 | 0.3298 | 0.3791 | 0.0203 | 0.3791 | 0.8523 | **0.8840** |
| | | R@10 | 0.4418 | 0.4976 | 0.0423 | 0.4724 | 0.8954 | **0.9328** |
| **Multi-API test data** | MRR | | 0.2296 | 0.2879 | 0.0355 | 0.2988 | **0.6495** | 0.5876 |
| | MAP | | 0.2212 | 0.2804 | 0.0115 | 0.2895 | **0.6392** | 0.5755 |
| | Precision | P@1 | 0.1280 | 0.1860 | 0.004 | 0.1970 | 0.6770 | **0.7200** |
| | | P@3 | 0.1166 | 0.1156 | 0.0073 | 0.1766 | 0.6489 | **0.7009** |
| | | P@5 | 0.1162 | 0.0850 | 0.018 | 0.1692 | 0.6365 | **0.6898** |
| | | P@10 | 0.1120 | 0.0542 | 0.0153 | 0.1585 | 0.6183 | **0.6647** |
| | Recall | R@1 | 0.1155 | 0.1703 | 0.0011 | 0.1800 | 0.6406 | **0.6846** |
| | | R@3 | 0.2440 | 0.3126 | 0.0065 | 0.3183 | 0.7806 | **0.8306** |
| | | R@5 | 0.3188 | 0.3795 | 0.0278 | 0.3891 | 0.8335 | **0.8758** |
| | | R@10 | 0.4443 | 0.4856 | 0.0472 | 0.4979 | 0.8793 | **0.9098** |

**Results**. Table 3 shows the result of CLEAR compared with the other baseline approaches at the class level. Overall, CLEAR outperforms other baselines on each of the three datasets. Among the three baselines, similar to method-level API recommendations, BIKER reports better performance than RACK and DeepAPI.

On BIKER test data, the recall@1 of CLEAR-complete is 80.95%, indicating that there is at least one right answer in the top three candidates in 80.95% cases. Comparing the CLEAR-complete model with RACK, the CLEAR-complete model outperforms RACK by 187.65% in MRR and 196.76% in MAP. In terms of precision and recall, CLEAR-complete improves the *precision*@1, 3, 5, 10 by 236.11%, 566.25%, 906.04%, 1684.76% and *recall*@1 by 242.86%, 158.87%, 144.96%, 129.77% respectively. Comparing the CLEAR-complete model with the BIKER-complete model, the CLEAR-complete model outperforms the BIKER-complete model by 7.70% in MRR and 9.43% in MAP. On the random test data, CLEAR outperforms RACK, BIKER, and DeepAPI in all the measurements. Comparing the CLEAR-complete model with RACK, CLEAR-complete outperforms RACK by 273.41% in MRR, 298.55% in MAP, 432.89% in precision@1, and 455.62% in Recall@1. On the multiple-API test data, CLEAR-complete model outperforms RACK by 187.33% in MRR, 197.04% in MAP. In terms of precision and recall, CLEAR-complete outperforms RACK the *precision*@ 1, 3, 5, 10 by 393.20%, 642.09%, 941.02%, 1617.85% and *recall*@1, 3, 5, 10 by 399.01%, 193.87%, 159.94%, 124.76% respectively. Compared to the RoBERTa model, CLEAR filter model achieves consistently better performance on each of the three datasets, indicating the effectiveness of contrastive learning.

The Wilcoxon signed-rank test ($p < 0.05$) also suggests that CLEAR achieves significantly better performance than all other baseline approaches.

> CLEAR significantly outperforms the state-of-the-art baselines at class-level API recommendation and CLEAR's performance remains stable across the three test datasets.

## 5.3 RQ3: Impact of Random Sampling

**Experimental Method**. In CLEAR's triplet generation, for queries with more than 10 positive or negative samples, CLEAR randomly selects 10 for each query. To understand how does random sampling affects the performance of CLEAR, we re-run the triplet generation 100 times. Please note that fine-tuning the model with full training triplets is very time-consuming so we perform this experiment on a subset of the training triplets containing 92k pairs (i.e., a quarter of the full training triplets).

**Result**. Table 4 shows the impact of random sampling on the performance of CLEAR measured by the Average Error and Coefficient of Variation (CV). As we can see from the table, the average error on MRR is 0.85%, indicating that the difference of MRR introduced by random sampling between different runs is 0.85% on

**Table 3: Performance comparison at class-level (RQ2)**

| Class-level | | | BIKER-filter | BIKER-complete | RACK | DeepAPI | RoBERTa | CLEAR-filter | CLEAR-complete |
|---|---|---|---|---|---|---|---|---|---|
| **BIKER test data** | MRR | | 0.6397 | 0.8138 | 0.3047 | 0.0172 | 0.5761 | 0.7059 | **0.8765** |
| | MAP | | 0.6343 | 0.8138 | 0.3001 | 0.0008 | 0.5769 | 0.7156 | **0.8906** |
| | Precision | P@1 | 0.2777 | 0.4642 | 0.2420 | 0.0044 | 0.3690 | 0.7777 | **0.8134** |
| | | P@3 | 0.2328 | 0.2486 | 0.1203 | 0.0014 | 0.4404 | 0.7513 | **0.8015** |
| | | P@5 | 0.2071 | 0.1698 | 0.0777 | 0.0079 | 0.4269 | 0.7380 | **0.7817** |
| | | P@10 | 0.1928 | 0.0956 | 0.0420 | 0.0070 | 0.4095 | 0.7182 | **0.7496** |
| | Recall | R@1 | 0.4623 | 0.6865 | 0.2361 | 0.0001 | 0.3611 | 0.5436 | **0.8095** |
| | | R@3 | 0.7559 | **0.9067** | 0.3472 | 0.0001 | 0.7202 | 0.8253 | 0.8988 |
| | | R@5 | 0.8531 | **0.9563** | 0.3750 | 0.0018 | 0.8214 | 0.8750 | 0.9186 |
| | | R@10 | 0.9424 | **0.9880** | 0.4067 | 0.0035 | 0.9226 | 0.8988 | 0.9345 |
| **Random test data** | MRR | | 0.4060 | 0.4515 | 0.2343 | 0.0206 | 0.4426 | 0.8467 | **0.8749** |
| | MAP | | 0.3961 | 0.4408 | 0.2207 | 0.0010 | 0.4410 | 0.8536 | **0.8796** |
| | Precision | P@1 | 0.1420 | 0.1740 | 0.1520 | 0.0060 | 0.3030 | 0.7800 | **0.8100** |
| | | P@3 | 0.1266 | 0.1103 | 0.0989 | 0.0026 | 0.2976 | 0.7719 | **0.8093** |
| | | P@5 | 0.1160 | 0.0830 | 0.0722 | 0.0076 | 0.2925 | 0.7611 | **0.7989** |
| | | P@10 | 0.1074 | 0.0546 | 0.0431 | 0.0085 | 0.2810 | 0.7400 | **0.7809** |
| | Recall | R@1 | 0.2473 | 0.3103 | 0.1395 | 0.0002 | 0.2833 | 0.7473 | **0.7751** |
| | | R@3 | 0.4573 | 0.4881 | 0.2728 | 0.0003 | 0.4988 | 0.8806 | **0.9113** |
| | | R@5 | 0.5568 | 0.5571 | 0.3308 | 0.0016 | 0.5908 | 0.9133 | **0.9403** |
| | | R@10 | 0.6633 | 0.6880 | 0.3968 | 0.0039 | 0.7063 | 0.9395 | **0.9606** |
| **Multi-API test data** | MRR | | 0.3829 | 0.4458 | 0.2511 | 0.0193 | 0.4351 | **0.7702** | 0.7215 |
| | MAP | | 0.3763 | 0.4371 | 0.2406 | 0.001 | 0.4288 | **0.7684** | 0.7147 |
| | Precision | P@1 | 0.1280 | 0.1860 | 0.1620 | 0.002 | 0.3040 | 0.7720 | **0.7990** |
| | | P@3 | 0.1166 | 0.1156 | 0.1069 | 0.0013 | 0.2843 | 0.7513 | **0.7933** |
| | | P@5 | 0.1162 | 0.0850 | 0.0758 | 0.0094 | 0.2803 | 0.7415 | **0.7891** |
| | | P@10 | 0.1120 | 0.0542 | 0.0448 | 0.0087 | 0.2698 | 0.7312 | **0.7696** |
| | Recall | R@1 | 0.2263 | 0.3048 | 0.1525 | 0.0077 | 0.2811 | 0.7340 | **0.7610** |
| | | R@3 | 0.4341 | 0.4901 | 0.3003 | 0.0016 | 0.4830 | 0.8458 | **0.8825** |
| | | R@5 | 0.5298 | 0.5620 | 0.3548 | 0.0024 | 0.5756 | 0.8853 | **0.9223** |
| | | R@10 | 0.6688 | 0.6668 | 0.4208 | 0.0045 | 0.6905 | 0.9226 | **0.9458** |

**Table 4: Impact the random sampling in triplet generation**

| Metric | MRR | MAP |
|---|---|---|
| Average Error | 0.85% | 0.84% |
| Coefficient of Variation (CV) | 0.004 | 0.011 |

average. The Coefficient of Variation is calculated by $CV = \sigma/\mu$ [8], where $\sigma$ is the standard deviation and $\mu$ is the mean. The CV of our result suggests that the difference introduced by random sampling is negligible.

The impact of random sampling in triplet generation on the performance of CLEAR is negligible, which shows the robustness of CLEAR.

## 6 DISCUSSIONS

This section discusses open questions regarding the performance and threads to validity of CLEAR .

### 6.1 Why CLEAR Outperforms Existing Baselines?

To understand why CLEAR significantly outperforms the baselines introduced in Section 4.4, we visualize the embedding of the API search space of the model before and after contrastive training. Specifically, we use the Uniform Manifold Approximation and Projection (UMAP) [18] approach to reduce the dimension of the BERT-based sentence embedding to two dimensions. Then we label the embedding vectors with the Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [3], an unsupervised cluster classification approach for the coloring.

Figure 5 shows the visualization, in which the upper graph shows the sentence embedding visualization of the training samples on the model before we apply contrastive training, in which the points represent the sentence embedding vectors in two-dimensional space and the color of the points indicates the APIs. From the visualization, we can see that the majority of the APIs are mixed and the boundary of each API is not clear. This graph shows clearly that it is very hard to draw the decision boundary for different clusters in the model before contrastive training. Since the training target of contrastive training is to minimize the distance between semantically equivalent sentences and maximize the distance between the irrelevant sentence, the margin between clusters should be larger and clearer after training.

To support the above hypothesis, we also apply the same visualization approach to the fine-tuned model after we applied contrastive training. Figure 5 lower graph shows the sentence embedding visualization of the training samples after contrastive training. From this figure, we can see clear cluster patterns of the query embedding vectors. Most of the APIs are from dense clusters and the margin space between clusters is relatively clear. This visualization supports our hypothesis of contrastive training, meaning that the contrastive training does pull semantic equivalent queries together and separates the irrelevant vectors apart.

**Table 5: Recommendation results (i.e., APIs and top similar questions) of CLEAR for example queries. ✓ indicates the ground-truth API and ✗ indicates the recommended API is incorrect.**

| | Question | API answers |
|---|---|---|
| input query | How to convert DateFormat "Fri Jan 08 13:48:16 GMT+05:30 2021" to java.sql.Date | ✓java.text.SimpleDateFormat.parse |
| 1st | **How to parse "Thu Aug 04 00:00:00 IST 2011" to "04-08-2011"?** | ✓java.text.SimpleDateFormat.parse |
| 2nd | Converting "2010-02-15T20:05:28.000Z" in GMT format using Java | ✓java.text.SimpleDateFormat.parse |
| 3rd | Convert String date into java.util.Date in the dd/MM/yyyy format | ✗java.text.DateFormat.format |
| 4th | Date format and the hour is always 12:00:00.000 | ✗java.time.Instant.parse |
| input query | How to retrive value from property file which are present outside of the app | ✓java.util.Properties.load |
| 1st | **How to close the fileInputStream while reading the property file** | ✓java.util.Properties.load |
| 2nd | Using Maven properties to connect to a database | ✗java.lang.System.getProperty |
| 3rd | Why do we need Properties class in java? | ✗java.util.Properties.load |
| 4th | Issue reading a file path from a Properties file | ✗java.util.Properties.store |

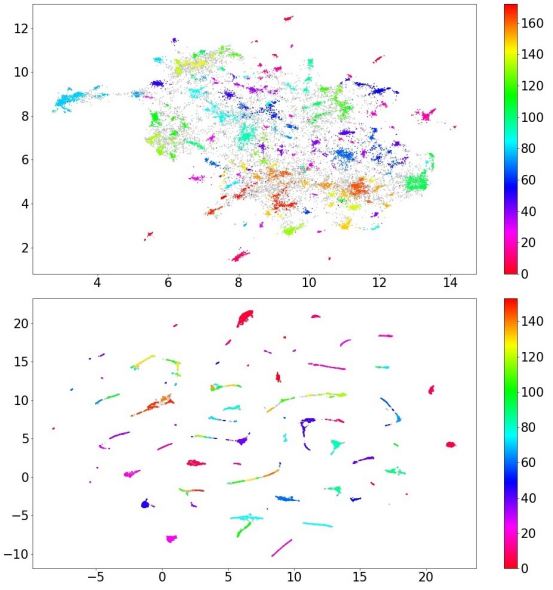

**Figure 5: Visualization of API question sentence embedding before (i.e., the upper image) and after (i.e., the lower image) contrastive training.**

## 6.2 CLEAR in the Real-world Practice

We run CLEAR, BIKER, RACK, and DeepAPI on 50 recent Java-related questions from Stack Overflow[9]. Comparing the top 10 recommended APIs, CLEAR successfully recommends APIs for 34 queries, BIKER successfully recommends APIs for 23 queries, RACK successfully recommends APIs for 4 queries, and DeepAPI successfully recommends APIs for 2 queries.

We selected two random examples that can be solved by CLEAR only for demonstration. Table 5 shows the recommendation results of CLEAR for the two example latest SO posts. The first example is about converting date formats, we can see that CLEAR can understand the concept of time in multiple formats and pick the keyword "convert" correctly. The result shows that CLEAR is not suffering from the lexical similarity pitfall concerning the time format and is able to recommend correct APIs. The second example is about property file access,the semantic of the question is "how to load property files" and the CLEAR is able to get the keywords that are the most related to the question, i.e., "property file" and "retrieve". We also see that the keyword "reading", the synonym of "retrieve", is correctly recognized as well.

Through the above two case studies, we can see that the CLEAR is more effective in capturing the semantic of the API queries regardless of the lexical information, thus can be used for API recommendation in a real-world application.

[9]*The full list can be found in the reproduction package

## 6.3 Threat to Validity

**Internal Validity**. Our code has been checked to ensure our implementation is correct and the questions in the testing dataset are not included in the question base. And we reuse the replication packages of the baselines to ensure their correctness. In addition, although the dataset collected from SO is being filtered by heuristic rules, there are still noises in the dataset due to the openness of SO, which may affect the performance of the CLEAR.

**External Validity**. In this work, we used the dataset published by BIKER to demonstrate the effectiveness of CLEAR, which only supports Java API recommendations. The performance of CLEAR can be different on API recommendation for other programming languages. In addition, as the dataset only contains questions from Stack Overflow, CLEAR might perform differently on data collected from other online forums. Future study is needed to examine the performance of CLEAR on data from other sources.

**Construct Validity** We use MRR, MAP, *precision@k*, and *recall@k* to measure the performance of API recommendation [14, 35], our approach might have different performance under other metrics. In this work, we assume that SO questions with the same API answer as semantically equivalent when contrastively training our BERT-based sentence embedding. Future study is needed to examine our assumption on API Q&A pairs from other sources or other tasks.

# 7 RELATED WORK

## 7.1 API Recommendation

There are many existing studies on API recommendation, including API invocation sequences mining [19], dependency graph-based API phrases mining [4], API recommendation for feature

requests [43], query-API keyword mapping with crowed knowledge [35], code snippet synthesis [33], similarity-based API recommendation with language model [14], and API recommendation based on similarity of functionality verb phrases in functionality descriptions and user queries [53].

McMillan et al. [19] first presented *portfolio*, an API recommendation tool that returns code snippets for a programming query. Thung et al. [43] introduced historical feature requests combined with official API documents information for API recommendation for new feature requests. Nguyen et al. [23] proposed GRALAN, a graph-based language model for object-oriented source codes. Liu et al. [16] improved the ranking of the top-10 result of GRALAN by introducing API usage path information to the graph system. Nguyen et al. [22] used statistical learning on the commit changes information for API recommendation. Gu et al. [11] first introduced a deep learning model to API learning which achieves end-to-end API sequence generation. CLEAR uses RoBERTa as the base model, which is different from DeepAPI. Rahman et al. [35] presented RACK, an API recommendation tool leveraging the real API usage data from Stack Overflow [26]. The difference between the RACK and CLEAR is that CLEAR uses a language model instead of keyword mapping. Huang et al. [14] proposed BIKER, which filters the candidate APIs based on the similarity against SO questions and then re-ranks the candidates based on the similarity against official API documentation description. The main difference between BIKER and CLEAR is that CLEAR uses contrastive training instead of unsupervised training in the model building stage.

### 7.2 API Usage Pattern Mining

Xie et al. [52] proposed MAPO, an API usage pattern mining tool with various code pattern mining algorithms. Thummalapenta et al. [42] proposed PARSEWeb, a java code reuse example generation tool build upon open-source java code data. Tseng et al. [44] proposed UP-miner, a toolset that contains thirteen java utility code pattern mining algorithms that improve the performance of UP-miner. Fowkes et al. [9] presented PAM, a parameter-free probabilistic algorithm for mining the API usage patterns. Wen et al. [49] proposed an API miss-use detection tool that can detect API misuse patterns of Java libraries. Chen et al. [5] first applied an unsupervised technique to create analogical API mappings of third-party libraries. Ren et al. [37] built an API-constraint knowledge graph for API-misuse detection purpose.

## 8 CONCLUSION

In this paper, we propose CLEAR, a novel approach for API recommendation. CLEAR uses the BERT-based model for embedding, which produces the embedding of the whole sentence of an API query while considering semantic-related sequential information. It uses contrastive training to better capture the semantics of the API queries regardless of the lexical information. Our experiment results confirm the effectiveness of the CLEAR for both methods- and class-level API recommendation. Our case study with CLEAR on the latest SO posts further demonstrates its practical value.

In the future, we plan to extend CLEAR to other tasks such as third-party API recommendation, Linux command search, code snippet search, and program patch search.

## 9 ACKNOWLEDGMENTS

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

## REFERENCES

[1] Laura Aina, Kristina Gulordava, and Gemma Boleda. 2019. Putting words in context: LSTM language models and lexical ambiguity. *arXiv preprint arXiv:1906.05149* (2019).
[2] Ekaba Bisong. 2019. Google colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 59–64.
[3] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 160–172.
[4] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[5] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* (2019).
[6] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. 2020. Big self-supervised models are strong semi-supervised learners. *arXiv preprint arXiv:2006.10029* (2020).
[7] 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).
[8] Brian Everitt and Anders Skrondal. 2002. *The Cambridge dictionary of statistics*. Vol. 106. Cambridge University Press Cambridge.
[9] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 254–265.
[10] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.
[11] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.
[12] Zellig S Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162.
[13] Elad Hoffer and Nir Ailon. 2015. Deep metric learning using triplet network. In *International workshop on similarity-based pattern recognition*. Springer, 84–92.
[14] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.
[15] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *arXiv preprint arXiv:2004.11362* (2020).
[16] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API recommendation without historical software repositories. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 282–292.
[17] 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).
[18] Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).
[19] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.
[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546* (2013).
[21] Jaechang Nam and Sunghun Kim. 2015. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 452–463.
[22] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th*

*ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.

[23] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 858–868.

[24] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. Focus: A recommender system for mining api function calls and usage patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1050–1060.

[25] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[26] Stack Overflow. 2008. https://stackoverflow.com/.

[27] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[28] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365* (2018).

[29] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668* (2018).

[30] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. 2014. Improving low quality stack overflow post detection. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 541–544.

[31] Dragomir R Radev, Hong Qi, Harris Wu, and Weiguo Fan. 2002. Evaluating Web-based Question Answering Systems.. In *LREC*. Citeseer.

[32] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

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

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

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

[36] Radim Řehřek, Petr Sojka, et al. 2011. Gensim—statistical semantics in python. *Retrieved from genism. org* (2011).

[37] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 461–472.

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

[39] Rodrigo Silva, Chanchal Roy, Mohammad Rahman, Kevin Schneider, Klerisson Paixao, and Marcelo Maia. 2019. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 358–368.

[40] Jake Snell, Kevin Swersky, and Richard S Zemel. 2017. Prototypical networks for few-shot learning. *arXiv preprint arXiv:1703.05175* (2017).

[41] Fadi Thabtah, Suhel Hammoud, Firuz Kamalov, and Amanda Gonsalves. 2020. Data imbalance in classification: Experimental evaluation. *Information Sciences* 513 (2020), 429–441.

[42] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 204–213.

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

[44] Vincent S Tseng, Cheng-Wei Wu, Jun-Han Lin, and Philippe Fournier-Viger. 2015. UP-miner: a utility pattern mining toolbox. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE, 1656–1659.

[45] EM Voorhees. 1999. Proceedings of the 8th text retrieval conference. *TREC-8 Question Answering Track Report* (1999), 77–82.

[46] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2019. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1259–1276.

[47] Song Wang, Chetan Bansal, and Nachiappan Nagappan. 2021. Large-scale intent analysis for identifying large-review-effort code changes. *Information and Software Technology* 130 (2021), 106408.

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

[49] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 866–877.

[50] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[51] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[52] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*. 54–57.

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