Effective Reformulation of Query for Code Search using Crowdsourced Knowledge and Extra-Large Data Analytics

***Abstract*—Software developers frequently issue generic natural language queries for code search while using code search engines (e.g., GitHub native search, Krugle). Such queries often do not lead to any relevant results due to vocabulary mismatch problems. In this paper, we propose a novel technique that automatically identifies relevant and specific API classes from Stack Overflow Q & A site for a programming task written as a natural language query, and then reformulates the query for improved code search. We first collect candidate API classes from Stack Overflow using pseudo-relevance feedback and two term weighting algorithms, and then rank the candidates using Borda count and semantic proximity between query keywords and the API classes. The semantic proximity has been determined by an analysis of 1.3 million questions and answers of Stack Overflow. Experiments using 310 code search queries report that our technique suggests relevant API classes with 48% precision and 58% recall which are 32% and 48% higher respectively than those of the state-of-the-art. Comparisons with two state-of-the-art studies and three popular search engines (e.g., Google, Stack Overflow, and GitHub native search) report that our reformulated queries (1) outperform the queries of the state-of-the-art, and (2) significantly improve the code search results provided by these contemporary search engines.**

I. Introduction

Software developers spend about 19% of their development time in searching for relevant code snippets (e.g., API usage examples) on the web [18]. Although open source software repositories (e.g., GitHub, SourceForge) are a great source of such code snippets, retrieving them is a major challenge [14]. Developers often use code search engines (e.g., Krugle, GitHub native search) to collect code snippets from such repositories using generic natural language queries [15]. Unfortunately, such queries hardly lead to any relevant results (i.e., only 12% [15]) due to vocabulary mismatch issues [26, 46]. Hence, the developers frequently reformulate their queries by removing irrelevant keywords and by adding more appropriate keywords. Studies [37, 67, 15] have shown that 33%–73% of all the queries are incrementally reformulated by the developers. These manual reformulations involve numerous trials and errors, and often cost significant development time and efforts [37]. One way to help the developers overcome this challenge is to automatically reformulate their generic queries (which are often poorly designed [37, 46]) using appropriate query keywords such as relevant API classes. Our work in the paper addresses this particular research problem – *query reformulation* targeting *code search*.

Several existing studies offer automatic query reformulation supports for code search using either actual or pseudo relevance feedback on the query [55, 75] and by mining crowd generated knowledge stored in Stack Overflow programming Q & A site [55, 43, 12]. Nie et al. [55] collect pseudo-relevance feedback (PRF) on a given query by employing Stack Overflow as a feedback provider, and then suggest query expansion by analysing the feedback documents, i.e., relevant programming questions and answers. However, they treat the Q & A threads as regular texts, and suggest natural language (i.e., software-specific) terms as query expansion. Existing evidence suggests that queries containing only natural language terms perform poorly in code search [15]. Rahman et al. [63] mine co-occurrences between query keywords (found in the question titles) and API classes (found in the answers) of Stack Overflow, apply two heuristics, and then suggest a set of relevant API classes for a given query. Unfortunately, their heuristics are likely to return highly *generic* and *frequent* API classes (e.g., `String, ArrayList, List`) due to their sole reliance on the co-occurrences. They even might return false positives given that all Q & A threads from the corpus were used for each query rather than the relevant ones.

In this paper, we propose a novel technique–NLP2API–that automatically identifies relevant API classes for a programming task written as a natural language query, and then reformulates the query using these API classes for improved code search. We first (1) collect candidate API classes for a query from relevant questions and answers of Stack Overflow (i.e., *crowdsourced knowledge*) (Section III-A), and then (2) identify appropriate classes from the candidates using Borda count (Section III-B) and query-API semantic proximity (i.e., *extra-large data analytics*) (Section III-C). In particular, we determine semantics of either a keyword or an API class based on their positions within a high dimensional semantic space developed by *fastText* [17] using 1.3 million questions and answers of Stack Overflow. Then we estimate the relevance of the candidate API class to the search query using their semantic proximity measure. Earlier approaches only perform either local context analysis [55, 75] or global context analysis [63, 47, 42]. On the contrary, our technique analyses both local (e.g., PageRank [19]) and global (e.g., semantic proximity) contexts of the query keywords for relevant API class identification and query reformulation. Thus, NLP2API has a higher potential for query reformulation. Besides, opportunistic blending of pseudo-relevance feedback [68, 21], term weighting methods [19, 35], Borda count [82] and *extra-large data analytics* [17] also makes our work in this paper *novel*.

Table 1 and Listing 1 present a use-case scenario of our technique where a developer is looking for a working

```
BufferedImage master = ImageIO.read(new URL(
"http://www.java2s.com/style/download.png"));
BufferedImage gray = new BufferedImage(master.getWidth(),
master.getHeight(), BufferedImage.TYPE_INT_ARGB);

ColorConvertOp op = new ColorConvertOp(
ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
op.filter(master, gray);

ImageIO.write(master,"png",new File("path/to/master"));
ImageIO.write(gray,"png", new File("path/to/gray/image"));
```

Listing 1. An example code snippet for the programming task– "Convert image to grayscale without losing transparency", (taken from [4])

TABLE I
REFORMULATIONS OF A NL QUERY FOR IMPROVED CODE SEARCH

Technique	Reformulated Query	QE
Baseline	Convert image to grayscale without losing transparency	115
QECK [55]	{Convert image grayscale losing transparency} + {hsb pixelsByte png iArray img correctly HSB mountainMap enhancedImagePixels file}	**11**
Google	Convert image to grayscale without losing transparency	**02**
Proposed	{Convert image grayscale losing transparency} + {**BufferedImage** Grayscale ImageEdit **ColorConvertOp** File Transparency **ColorSpace** BufferedImageOp Graphics ImageEffects}	**02**

QE = Rank of the first correct result returned by the query

code snippet that can convert a colour image to grayscale without losing transparency. First, the developer issues a generic query–*"Convert image to grayscale without losing transparency"*. Then she submits it to *Lucene*, a search engine that is widely used both by contemporary code search solutions such as GitHub native search [5] or ElasticSearch and by academic studies [31, 61, 54]. Unfortunately, the generic natural language query does not perform well due to vocabulary mismatch between its keywords and the source code, and returns the relevant code snippet (e.g., Listing 1) at the $\mathbf{115}^{th}$ position. On the contrary, (1) our proposed technique *complements* this query with not only *relevant*, but also highly *specific* API classes (e.g., **`BufferedImage`**, **`ColorConvertOp`**, **`ColorSpace`**), and (2) our improved query returns the target code snippet at the ***second*** position of the ranked list which is a major rank improvement over the baseline. The most recent and closely related technique–QECK [55] returns the same code snippet at the 11^{th} position which is not ideal. Google, the most popular web search engine, returns a *similar* code at the second position as well. However, in the case of web search, relevant code snippets are *sporadic* and often buried into a large bulk of unstructured, noisy and redundant natural language texts across multiple web pages which might overwhelm the developer with information overload [50].

Experiments using 310 code search queries randomly collected from four Java tutorial sites–*KodeJava, Java2s, CodeJava* and *JavaDB*–report that our technique can suggest relevant API classes with 82% Top-10 Accuracy, 48% precision, 58% recall and a reciprocal rank of 0.55 which are 6%, 32%, 48% and 41% higher respectively than those of the state-of-the-art [63]. Comparisons with two state-of-the-art studies and three popular code (or web) search engines – *Google*, *Stack Overflow native search* and *GitHub native search* – reported that our technique (1) can outperform the existing studies [55, 70] in query effectiveness and (2) can improve upon the precision of these search engines by 21%, 35% and 16% respectively using our reformulated queries. Thus, this paper makes the following contributions:

- A novel technique that reformulates generic natural language queries for improved code search using large data analytics and crowd knowledge stored in Stack Overflow.
- Comprehensive evaluation of the proposed technique using 310 queries and validation against the state-of-the-art techniques and widely used code search engines.
- A replication package that includes our working prototype and the detail experimental dataset.

II. BACKGROUND

Pseudo-Relevance Feedback (PRF): Gay et al. [27] first use relevance feedback for query reformulation in the context of Software Engineering, more specifically in concept location. They first collect a developer's feedback on a given query where the developer marks each result returned by the query as either *relevant* or *irrelevant*. Then they analyse the marked results, extract appropriate keywords using Rocchio's method [66], and reformulate the given query. Although the interactive feedbacks from the developer are effective, collecting them is time-consuming and sometimes infeasible [48]. Hence, researchers adopted a less effective but more feasible approach namely *pseudo-relevance feedback* [31, 55, 60]. In this approach, only the Top-K results (returned by the given query) are assumed as relevant and thus, are *automatically* analysed for the reformulation task. Studies have shown that pseudo-relevance feedback based reformulations could improve the initial queries significantly (i.e., $\approx$ 20%) [21]. In our work, we adopt pseudo-relevance feedback as a step for candidate API class selection from Stack Overflow Q & A threads.

Word Embeddings (WE): Traditional code search engines often suffer from vocabulary mismatch issues (e.g., polysemy, synonymy) [46, 79, 30]. One crucial step to overcome this challenge is to determine semantics of a word correctly. There have been several attempts [53, 81, 80, 59] to define the semantics of a word by using its contexts captured from a large corpus (e.g., Stack Overflow). Mikolov et al. [53] propose *word2vec*, a feed-forward neural network based text mining tool that mines a corpus, and represents each word as a point within a high-dimensional semantic space. Thus, the semantics of each word is represented as a vector, and similar words occur close to each other on the semantic space. Such vector is also called *word embeddings* [80, 53]. *word2vec* is trained using two predictive models– continuous bag of words (CBOW) and skip-gram. CBOW model predicts a word given its contextual words whereas skip-gram attempts to predict the context of a given word probabilistically. We use a faster version of *word2vec* called *fastText* [17] with its default parameters for query reformulation in this paper.

III. NLP2API: PROPOSED TECHNIQUE FOR QUERY REFORMULATION FOR IMPROVED CODE SEARCH

Fig. 1 shows the schematic diagram of our proposed technique for the reformulation of a generic query targeting code

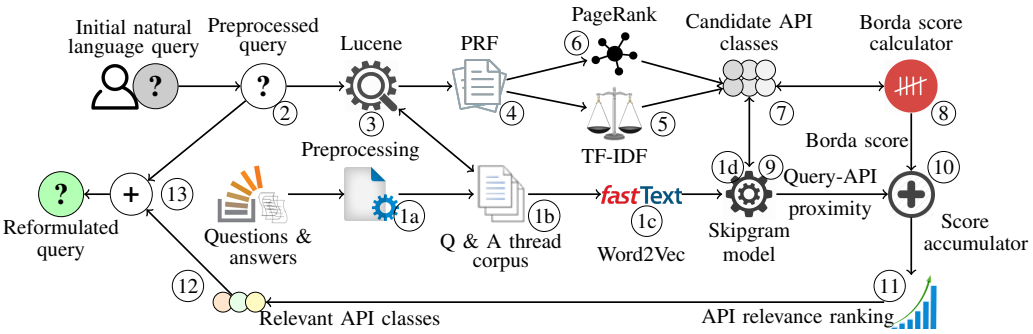

Fig. 1. Schematic diagram of the proposed query reformulation technique–NLP2API

search. Furthermore, Algorithm 1 shows the pseudo-code of our technique. We make use of pseudo-relevance feedback (PRF), crowd generated knowledge stored at Stack Overflow, two term weighting algorithms, and extra-large data analytics for our query reformulation as follows:

A. Development of Candidate API Lists

We collect candidate API classes from Stack Overflow Q & A site to reformulate a generic query (i.e., Fig. 1, Steps 1a, 1b, 2–7). Stack Overflow is a large body of crowd knowledge with 14 million questions and 22 million answers across multiple programming languages and domains [20]. Hence, it might contain at least a few questions and answers related to any programming task at hand. Earlier studies from the literature [55, 43, 20] also strongly support this conjecture. Given that relevant program elements are a better choice than generic natural language terms for code search [15], we collect API classes as candidates for query reformulation by mining the programming Q & A threads of Stack Overflow.

Corpus Preparation: We collect a total of 656,535 Q & A threads related to `Java` (i.e., using `<java>` tag) from Stack Overflow for corpus preparation (Fig. 1, Steps 1a, 1b, Algorithm 1, Line 3). We use the public data dump [13] released on March 2018 for data collection. Since we are mostly interested in the API classes discussed in the Q & A texts, we adopt certain restrictions. First, we make sure that each question or answer contains a bit of code, i.e., the thread is about coding. For this, we check the existence of `<code>` tags in their texts like the earlier studies [57, 25, 62, 24]. Second, to ensure high quality content, we chose only such Q & A threads where the answer was accepted as solution by the person who submitted the question [55, 63]. Once the Q & A threads are collected, we perform standard natural language preprocessing (i.e., removal of stop words, punctuation marks and programming keywords, token splitting) on each thread, and normalize their contents. Given the controversial evidence on the impact of stemming on source code [33], we avoid stemming on these threads given that they contain code segments. Our corpus is then indexed using *Lucene*, a widely used search engine by the literature [54, 31, 61], and later used for collecting feedbacks on a generic natural language query.

Pseudo-Relevance Feedback (PRF) on the NL Query: Nie et al. [55] first employ Stack Overflow in collecting pseudo-relevance feedback on a given query. Their idea was to extract software-specific words relevant to a given query, and then to use them for query reformulation. Similarly, we also collect pseudo-relevance feedback on the query using Stack Overflow. We first normalize a natural language query using standard natural language preprocessing (i.e., stopword removal, token splitting), and then use it to retrieve Top-M (e.g., $M = 35$) Q & A threads from the above corpus with *Lucene* search engine (i.e., Fig. 1, Steps 2–4, Algorithm 1, Lines 4–8). The baseline idea is to extract appropriate API classes from them using appropriate selection methods [45], and then, to use them for query reformulation. We thus extract the program elements (e.g., code segments, API classes) from each of the threads by analysing their HTML contents. We use *Jsoup* [8], a Java library for the HTML scraping. We also develop *two* separate sets of code segments from the questions and answers of the feedback threads. Then we use *two* widely used term-weighting methods *–TF-IDF* and *PageRank–* for collecting candidate API classes from them.

API Class Weight Estimation with TF-IDF: Existing studies [31, 55, 27] often apply Rocchio's method [66] for query reformulation where they use TF-IDF to select appropriate expansion terms. Similarly, we adopt TF-IDF for selecting potential reformulation candidates from the code segments that were collected above. In particular, we extract all API classes from each code segment (i.e., feedback document) with the help of island parsing (i.e., uses regular expressions) [65], and then determine their relative weight (i.e., Fig. 1, Step 5, Algorithm 1, Lines 11–12) as follows:

$$TF - IDF(A_i) = (1 + log(TF_{A_i})) \times log(1 + \frac{N}{DF_{A_i}})$$

Here TF_{A_i} refers to total occurrence frequency of an API class A_i in the collected code segments, N refers to total Q & A threads in the corpus, and DF_{A_i} is the number of threads that mentioned API class A_i in their texts.

API Class Weight Estimation with PageRank: Semantics of a term are often determined by its contexts, i.e., surrounding terms [80, 81]. Hence, inter-dependence of terms is an important factor in the estimation of term weight. However, TF-IDF assumes term independence (i.e., ignores term contexts) in the weight estimation. Hence, it might fail to identify highly important but not so frequent terms from a body of texts [60, 52]. We thus employ another term weighting method that considers dependencies among terms in the weight estimation. In particular, we apply PageRank algorithm [19, 52] to the

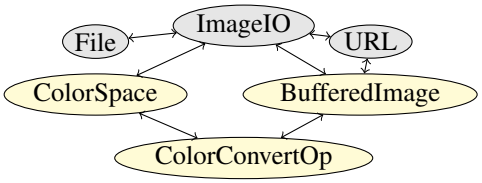

Fig. 2. API co-occurrence graph for code segment in Listing 1

relevance feedback documents, i.e., relevant code segments, and identify the important API classes as follows:

Construction of API Co-occurrence Graph: Since PageRank algorithm operates on a graph-based structure, we transform pseudo-relevance feedback documents into a graph of API classes (i.e., Fig. 1, Step 6, Algorithm 1, Line 13). In particular, we extract all API classes from each code segment using island parsing [65], and then develop an ordered list by preserving their initialization order in the code. For example, the code snippet in Listing 1 is converted into a list of six API classes. Co-occurrences of items in a certain context has long been considered as an indication of relatedness among the items [81, 52]. We thus capture the immediate co-occurrences of API classes in the above list, consider such co-occurrences as connecting edges, and then develop an API co-occurrence graph (e.g., Fig. 2). We repeat the same step for each of the code segments, and update the connectivities in the graph. We develop one graph for the code segments from questions and another graph for the code segments from answers which were returned as a part of the pseudo-relevance feedback.

API Class Rank Calculation: PageRank has been widely used for web link analysis [19] and term weighting in Information Retrieval domain [52]. It applies the underlying mechanism of recommendation or voting for determining importance of an item (e.g., web page, term) [61]. That is, PageRank considers a node as important only if it is recommended (i.e., connected to) by other important nodes in the graph. The same idea has been widely used for separating legitimate pages from spam pages [56]. Similarly, in our problem context, if an API class co-occurs with other important API classes across multiple code segments that are relevant to a programming task, then this API class is also considered to be important for the task. We apply PageRank algorithm on each of the two graphs (i.e., Fig. 1, Step 6, Algorithm 1, Line 14), and determine the importance $ACR(v_i)$ (i.e., API Class Rank) of each node v_i by recursively applying the following equation:

$$ACR(v_i) = (1-\phi) + \phi \sum_{j \epsilon In(v_i)} \frac{ACR(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1)$$

Here, $In(v_i)$ refers to nodes providing inbound links (i.e., votes) to node v_i whereas $Out(v_i)$ refers to nodes that v_i is connected to through outbound links, and ϕ is the damping factor. In the context of world wide web, Brin and Page [19] considered ϕ as the probability of visiting a web page and $1-\phi$ as the probability of jumping off the page by a random surfer. We use a value $\phi = 0.85$ for our work like the previous studies [19, 52, 60]. We initialize each node with a score of 0.25, and run an iterative version of PageRank on the graph. The algorithm pulls out weights from the surrounding nodes recursively, and updates the weight of a target node. This recursive process continues until the scores of the nodes converge below a certain threshold (e.g., 0.0001 [52]) or total iteration count reaches the maximum (e.g., 100 [52]). Once the computation is over, each node (i.e., API class) is left with a score which is considered as a numerical proxy to its relative importance among all nodes.

Algorithm 1 Query Reformulation using Relevant API Classes

```
1:  procedure NLP2API(Q)                          ▷ Q: natural language query
2:      R ← {}                                    ▷ R: Relevant API classes
3:      C ←developQ&ACorpus(SODump)               ▷ C: SO corpus
4:      Q_pp ←preprocess(Q)
5:      ▷ collecting pseudo-relevance feedback
6:      PRF ←getPRF(Q_pp, C)
7:      PRF_Q ← getQuestionCodeSegments(PRF)
8:      PRF_A ← getAnswerCodeSegments(PRF)
9:      ▷ collecting candidate API list
10:     for PRF prf ∈ {PRF_Q, PRF_A} do
11:         TW ←calculateTFIDF(prf, C)
12:         WC[prf] ←getTopKWeightedClasses(TW)
13:         G ←developAPICo-occurrenceGraph(prf)
14:         ACR ←calculateAPIClassRank(G)
15:         RC[prf] ←getTopKRankedClasses(ACR)
16:     end for
17:     ▷ training the fastText model
18:     M_ft ←getFastTextModel(preprocess(SODump))
19:     ▷ API relevance estimation
20:     A ←getAllCandidateAPIClasses(RC ∪ WC)
21:     for CandidateAPIClass A_i ∈ A do
22:         ▷ calculate Borda score
23:         S_B[A_i] ← getBordaScore(A_i, RC, WC)
24:         ▷ semantic relevance between API class and query
25:         S_P[A_i] ← getQuery-APIProximity(A_i, Q_pp, M_ft)
26:         R[A_i].score ← S_B[A_i] + S_P[A_i]
27:     end for
28:     ▷ ranking of the API classes
29:     rankedClasses ← sortByFinalScore(R)
30:     ▷ reformulation of the initial query
31:     return Q_pp + rankedClasses
32: end procedure
```

Selection of Candidate API Classes: Once two weights – TF-IDF and PageRank– of each of the potential candidates are calculated, we rank the candidates according to their weights. Then we select Top-N (e.g., $N = 16$) API classes from each of the four lists (i.e., two lists for each term weight, Fig. 1, Step 7, Algorithm 1, Lines 9–16). In Stack Overflow Q & A site, a question often describes a programming problem (or a task) whereas the answer offers a solution. Thus, API classes involved with the problem and API classes forming the solution should be treated differently for identifying the *relevant* and *specific* API classes for the task. We leverage this inherent differences of context and semantics between questions and answers, and treat their code segments separately unlike the earlier study of Nie et al. [55] that overlooks such differences.

B. Borda Score Calculation

Borda count is a widely used election method where the voters sort their political candidates on a scale of preference [82, 1]. In the context of Software Engineering, Holmes and

Murphy [34] first apply Borda count to recommend relevant code examples for the code under development in the IDE. They apply this method to six ranked list of code examples collected using six structural heuristics, and then suggest the most frequent examples across these lists as the most relevant ones. Similarly, we apply this method to our four candidate API lists (i.e., Fig. 1, Step 8, Algorithm 1, Lines 22–23) where each of the API classes are ranked based on their importance estimates (e.g., TF-IDF, API Class Rank). We calculate Borda score S_B for each of the API classes ($\forall A_i \in A$) from the these ranked candidate lists–$WRC = \{WC_Q, WC_A, RC_Q, RC_A\}$– as follows:

$$S_B(A_i \in A) = \sum_{RL_j \in WRC} 1 - \frac{rank(A_i, RL_j)}{|RL_j|}$$

Here, A refers to the set of all API classes extracted from the ranked candidate lists –WRC, $|RL_j|$ denotes each list size, and $rank(A_i, RL_j)$ returns the rank of class A_i in the ranked list. Thus, an API class that occurs at the top positions in multiple candidate lists is likely to be more important for a target programming task than the ones that either occurs at the lower positions or does not occur in multiple lists.

C. Query-API Semantic Proximity Analysis

Pseudo-relevance feedback, PageRank (Section III-A) and Borda count (Section III-B) analyse local contexts of the query keywords within a set of tentatively relevant documents (i.e., Q & A threads) and then extract candidate API classes for query reformulation. Although local context analysis is useful, existing studies report that such analysis alone might cause topic drift from the original query [43, 21]. We thus further analyse global contexts of the query keywords, and determine the semantic proximity between the given natural language query and the candidate API classes as follows:

Word2Vec Model Development: Mikolov et al. [53] and colleagues propose a neural network based tool–*word2vec*– for learning word embeddings from an ultra-large body of texts where they employ continuous bag of words (CBOW) and skip-gram models. While other studies attempt to define context of a word using co-occurrence frequencies or TF-IDF [81, 63, 49], they offer a probabilistic representation of the context. In particular, they learn *word embeddings* (Section II) for each of the words from the corpus, and map each word to a point in the semantic space so that semantically similar words appear in the close proximity. We leverage this notion of *semantic proximity*, and determine the relevance of a candidate API class to the given query. It should be noted that such proximity measure could be an effective tool to overcome the *vocabulary mismatch issues* [26]. We thus develop a *word2vec* model where 1.3 million programming questions and answers (i.e., 656,535 Q & A pairs, collected in Section III-A) are employed as the corpus. We normalize each question and answer using standard natural language preprocessing, and learn the word embeddings (Fig. 1, Step 1b, 1c, 1d, Algorithm 1, Lines 17–18) using skip-gram model. For our learning, we use *fastText* [17], an improved version of *word2vec* that incorporates sub-word information into the model. We performed the learning offline and it took about one hour. It should be noted that our model is learned using default parameters (e.g., output vector size = 100, context window size = 5, minimum word occurrences = 5) provided by the tool.

Semantic Relevance Estimation: While a given query contains multiple keywords, a candidate API class might not be semantically close to all of them. We thus capture the maximum proximity estimate between an API class and any of the query keywords as the relevance estimate of the class. In particular, we collect word embeddings (i.e., a vector of 100 probabilistic estimates of the contexts) of each candidate API class $A_i \in A$ and each keyword $q \in Q$, and determine their semantic proximity S_P using *cosine similarity* (i.e., Fig. 1, Step 9, Algorithm 1, Lines 24–25) as follows:

$$S_P(A_i \in A) = \{f(A_i, q) \mid f(A_i, q) > f(A_i, q_0) \forall q_0 \in Q\}$$
$$f(A_i, q) = cosine(fastText(A_i), fastText(q))$$

Here $fastText(.)$ returns the learned word embeddings of either a query keyword or an API class, and $f(A_i, q)$ returns the *cosine similarity* between their word embeddings. We use `print-word-vectors` command of *fastText* to collect the word embeddings from our learned model on Stack Overflow.

D. API Class Relevance Ranking & Query Reformulation

Once Borda score S_B and semantic proximity score S_P are calculated, we normalize both scores between 0 and 1, and then sum them up using a *linear combination* (i.e., Line 26, Algorithm 1) for each of the candidate API classes. While fine tuned relative weight estimation for these two scores could have been a better approach, we keep that as a part of future work. Besides, equal weights also reported pretty good results (e.g., 82% Top-10 accuracy) according to our investigation. The API classes are then ranked according to their final scores, and Top-K (e.g., $K = 10$) classes are suggested as the relevant classes for the programming task stated as a generic query (i.e., Fig. 1, Steps 10–12, Algorithm 1, Lines 19–29). These API classes are then appended to the given query as reformulations [31] (i.e., Fig. 1, Steps 13, Algorithm 1, Lines 30–31). Table I shows our reformulated query for the showcase natural language query using the suggested API classes.

IV. Experiment

We conduct experiments with 310 code search queries randomly collected from four popular programming tutorial sites, and evaluate our query reformulation technique. We choose five appropriate performance metrics from the literature, and evaluate two aspects of our provided supports-(1) relevant API class suggestion and (2) query reformulation. Our technique is also validated against two state-of-the-art techniques [63, 55] and three popular search engines. We thus answer five research question using our experiments as follows:

- **RQ_1:** How does NLP2API perform in recommending relevant API classes for a given query? How do different parameters and thresholds influence the performance?
- **RQ_2:** Can NLP2API outperform the state-of-the-art technique on relevant API class suggestion for a query?

- **RQ$_3$:** Can NLP2API improve the given natural language queries significantly using its suggested API classes?
- **RQ$_4$:** Can NLP2API outperform the state-of-the-art technique on query reformulation that uses crowdsourced knowledge from Stack Overflow?
- **RQ$_5$:** Can NLP2API significantly improve the results provided by state-of-the-art code or web search engines?

A. Experimental Dataset

Dataset Collection: We collect 310 code search queries from four popular programming tutorial sites–KodeJava [9], Java2s [6], CodeJava [2] and JavaDB [7]–for our experiments. While 150 of these queries were taken from a publicly available dataset [63], we further extended the dataset with 160 new queries. Each of these sites above discusses hundreds of programming tasks as Q & A threads where each thread generally contains (1) a question title, (2) a solution (i.e., code), and (3) a prose explaining the code succinctly. The question title (e.g., *"How do I decompress a GZip file in Java?"* [11]) generally comprises of a few important keywords and often resembles a real life search query. We thus use these titles as code search queries in our experiments, as were also used by the earlier studies from related literature [63, 22].

Ground Truth Preparation: The prose that explains code in the tutorial sites above often includes one or more API classes from the code (e.g., `GZipInputStream`, `FileOutputStream`). Since these API classes are chosen to explain the code that implements a programming task, they are generally relevant and specific to the task. We thus consider these *relevant* and *specific* API classes as the *ground truth* for the corresponding question title (i.e., our search query) [63]. We develop a *ground truth API set* to evaluate the performance of our technique in the API class suggestion. We also collect the code segments from each of the 310 Q & A threads from the tutorial sites above as the *ground truth code segments*, and use them to evaluate the query reformulation performance (i.e., in terms of code retrieval) of our technique. Given that these API classes and code segments are publicly available online and were consulted by thousands of technical users over the years, subjectivity associated with their relevance to the corresponding tasks (i.e., our selected queries) is minimized [22]. Our dataset preparation step took 25 man hours.

Replication Package: Our experimental data, working prototype and other materials are publicly available [10] for replication and third party reuse.

B. Performance Metrics

We choose five performance metrics that were widely adopted by relevant literature [51, 22, 74, 55, 63, 61], for the evaluation and validation of our technique as follows:

Top-K Accuracy / Hit@K: It is the percentage of search queries for each of which at least one item (e.g., API class) from the *ground truth* is returned within the Top-K results.

Mean Reciprocal Rank@K (MRR@K): Reciprocal Rank@K is defined as the multiplicative inverse of the rank of first relevant item (e.g., API class from ground truth) in the Top-K results returned by a technique. Mean Reciprocal Rank@K (MRR@K) averages such measures for all queries.

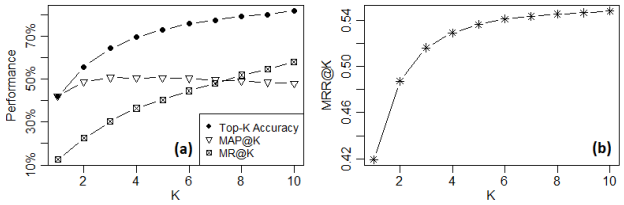

Fig. 3. Performance of NLP2API in API class suggestion for various Top-K results

TABLE II
PERFORMANCE OF NLP2API IN RELEVANT API SUGGESTION

Performance Metric	Top-1	Top-3	Top-5	Top-10
Top-K Accuracy	41.94%	64.19%	**72.90%**	**81.61%**
Mean Reciprocal Rank@K	0.42	0.52	**0.54**	**0.55**
Mean Average Precision@K	41.94%	**50.62%**	**50.56%**	47.85%
Mean Recall@K	12.53%	30.17%	**40.28%**	**57.87%**

Top-K = Performance measures for Top-K suggestions

Mean Average Precision@K (MAP@K): Precision@K is the precision calculated at the occurrence of Kth item in the ranked list. Average Precision@K (AP@K) averages the precision@K for all relevant items (e.g., API class from ground truth) within the Top-K results for a search query. Mean Average Precision@K is the mean of Average Precision@K for all queries from the dataset.

Mean Recall@K (MR@K): Recall@K is defined as the percentage of ground truth items (e.g., API classes) that are correctly recommended for a query in the Top-K results by a technique. Mean Recall@K (MR@K) averages such measures for all queries from the dataset.

Query Effectiveness (QE): It is defined as the rank of first correct item (i.e., ground truth code segment) in the result list returned by a query. The measure is an approximation of the developer's effort in locating the first code segment relevant to a given query. Thus, the lower the effectiveness measure is, the more effective the query is [61, 54]. We use this measure to evaluate the improvement of a query through reformulations offered by a technique.

C. Evaluation of NLP2API: Relevant API Class Suggestion

We first evaluate the performance of our technique in the relevant API class suggestion for a generic code search query. We make use of 310 code search queries (Section IV-A) and four performance metrics (Section IV-B) for this experiment. We collect Top-K (e.g., K=10) API classes suggested for each query, compare them with the *ground truth API classes*, and then determine our API suggestion performance. In this section, we also answer RQ$_1$ and RQ$_2$ as follows:

Answering RQ$_1$–Relevant API Class Suggestion: From Table II, we see that our technique returns relevant API classes for 73% of the queries with 51% mean average precision and 40% recall when only Top-5 results are considered. That is, half of the suggested classes come from the *ground truth*, and our approach succeeds for seven out of 10 queries. More importantly, it achieves a mean reciprocal rank of 0.54. That means, on average, the first relevant API class can be found at the second position of the result list. Such classes can also

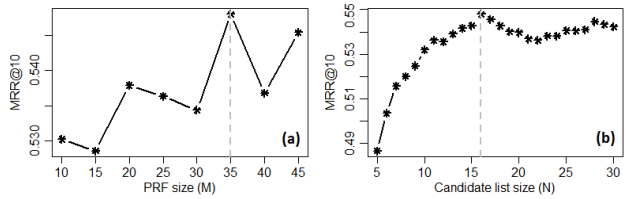

Fig. 4. Impact of (a) PRF size (M), and (b) Candidate API list size (N) on relevant API class suggestion from Stack Overflow

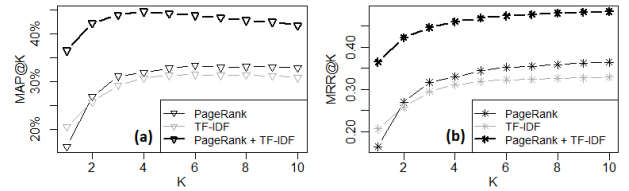

Fig. 5. Comparison between PageRank and TF-IDF in candidate API class selection using (a) MAP@K and (b) MRR@K

be found at the first position for 42% of the queries. All these statistics are highly promising according to relevant literature [74, 63]. Fig. 3 further demonstrates our performance measures for Top-1 to Top-10 results. We see that accuracy, recall and reciprocal rank measures increase monotonically which are expected. Interestingly, the precision measure shows an almost steady behaviour. That means, as more results were collected, our technique was able to *filter out* the *false positives* which demonstrates its high potential for API suggestion.

Impact of Pseudo-Relevance Feedback Size (M) and Candidate API List Size (N): We investigate how different sizes of pseudo-relevance feedback (i.e., number of Q & A threads retrieved from Stack Overflow by the given query) and candidate API list (i.e., detailed in Section III-A) affect the performance of our technique. We conduct experiments using 10–45 feedback Q & A threads and 5–30 candidate API classes. We found that these parameters improved accuracy and recall measures monotonically (i.e., as expected) but affected precision measures in an irregular fashion (i.e., not monotonic). However, we found an interesting pattern with mean reciprocal rank. From Fig. 4, we see that mean reciprocal rank@10 of our technique reaches the maximum when (a) pseudo-relevance feedback size, M is 35 and (b) candidate API list size, N is 16. We thus adopt these thresholds, i.e., $M = 35$ and $N = 16$, in our technique for the experiments.

Impact of Candidate Selection Method on the Overall Performance: We contrast between PageRank and TF-IDF by investigating their performances in the identification of candidate API classes from Stack Overflow. From Fig. 5, we see that our technique performs comparatively higher in terms of (a) precision and (b) reciprocal rank when PageRank is employed as opposed to TF-IDF for candidate API selection. However, when both PageRank and TF-IDF are employed, our technique performs significantly higher. We performed non-parametric statistical tests (e.g., *Mann-Whitney Wilcoxon*, *Cliff's Delta*) for significance. We found that precisions of our technique with PageRank+TF-IDF are significantly higher than those with either PageRank (i.e., *p-value*$\leq$0.001, $\Delta = 0.90$ (*large*)) or TF-IDF (i.e., *p-value*$\leq$0.001, $\Delta = 0.90$ (*large*)), which justifies our combination of these two selection methods.

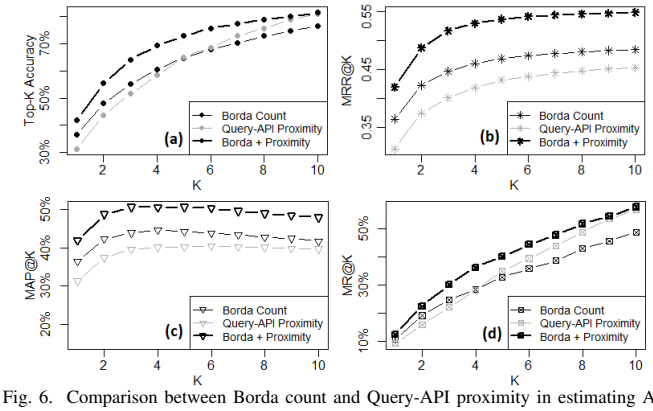

Fig. 6. Comparison between Borda count and Query-API proximity in estimating API relevance using (a) accuracy, (b) reciprocal rank, (c) precision, and (d) recall

TABLE III
COMPARISON WITH THE STATE-OF-THE-ART IN API CLASS SUGGESTION

Technique	Metric	Top-1	Top-3	Top-5	Top-10
RACK [63]	Top-K Accuracy	20.97%	52.90%	64.19%	77.10%
	MRR@K	0.21	0.35	0.37	0.39
	MAP@K	20.97%	34.76%	36.76%	36.38%
	MR@K	6.25%	20.81%	28.06%	39.22%
NLP2API (Proposed)	Top-K Accuracy	41.94%	64.19%	**72.90%**	**81.61%**
	MRR@K	0.42	0.52	**0.54**	**0.55**
	MAP@K	41.94%	**50.62%**	**50.56%**	47.85%
	MR@K	12.53%	30.17%	**40.28%**	**57.87%**

Top-K = Performance measures for Top-K suggestions

Borda Count vs. Query-API Class Proximity as API Relevance Estimate: Once candidate API classes are selected (Section III-A), we employ two proxies (Sections III-B, III-C) for estimating the relevance of an API class to the NL query. We compare the appropriateness of these proxies– *Borda Count* and *Query-API Proximity*– in capturing the API class relevance, and report our findings in Fig. 6. We see that Borda Count is more effective than Query-API Proximity in capturing the relevance of an API class to a given query. However, the proximity demonstrates its potential especially with accuracy and recall measures. More interestingly, combination of these two proxies ensures the best performance of our technique in all four metrics. Non-parametric statistical tests also report that performances with Borda+Proximity are significantly higher than those with either Borda Count (i.e., all *p-values*$<$*0.05*, $0.34 \leq \Delta \leq 0.82$ (*large*)) or Query-API Semantic Proximity (i.e., all *p-values*$<$*0.05*, $0.20 \leq \Delta \leq 0.90$ (*large*)).

We also investigate the parameters of *fastText* [17] that were used to determine the query-API proximity. Although we experimented using various custom parameters, we did not see any significant performance gain over the default parameters. Besides, increased thresholds (e.g., context window size, output vector size) could be computationally costly. We thus adopt the default settings of *fastText* in this work.

Our technique provides the first relevant API class at the *second* position, $\approx$ **50**% of our suggested classes are true positive, and the technique succeeds *eight* out of 10 times. Besides, our adopted parameters and thresholds are justified.

Answering RQ$_2$– Comparison with Existing Studies on Relevant API Class Suggestion: We compare our technique with the state-of-the-art approach – RACK [63] – on API class suggestion for a natural language query. Rahman et al. [63]

TABLE IV
IMPACT OF REFORMULATIONS ON GENERIC NL QUERIES

Reformulation	RL	Improved/MRD	Worsened/MRD	Preserved
NLP2API$_B$	05	43.23%/-245	31.29%/+54	25.48%
	10	**48.07**%/-223	26.13%/+65	25.81%
NLP2API$_P$	10	40.97%/-148	30.97%/+44	28.06%
NLP2API	05	40.00%/-159	27.74%/+54	32.26%
	10	**48.07**%/-209	25.16%/+45	**26.77**%
	15	**49.03**%/-217	**22.26**%/+46	**28.71**%

MRD = Mean Rank Difference between reformulated and given queries

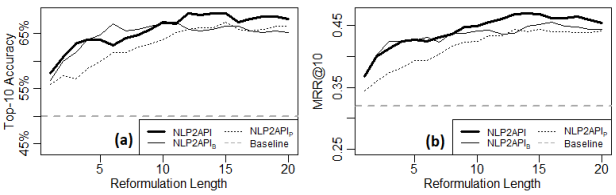

Fig. 7. Reformulated vs. baseline query using (a) Top-10 accuracy and (b) MRR@10

employ two heuristics– Keyword-API Co-occurrence (KAC) and Keyword-Keyword Coherence (KKC)–for suggesting relevant API classes from Q & A threads of Stack Overflow for a given query. Their approach outperformed earlier approaches [74, 22] which made it the state-of-the-art in relevant API class suggestion. We collect the authors' implementation of RACK from corresponding web portal, ran the tool as is on our dataset, and then extract the evaluation results.

From Table III, we see that our technique–NLP2API–outperforms RACK especially in precision, recall and reciprocal rank. It should be noted that our reported performance measures for RACK are pretty close to the authors' reported measures [63], which indicates a *fair comparison*. We see that RACK recommends API classes correctly for 64% of the queries with 37% precision, 28% recall and a reciprocal rank of 0.37 when Top-5 results are considered. On the contrary, our technique recommends correctly for 73% of the queries with 51% precision, 40% recall and a promising reciprocal rank of 0.54 in the same context. These are 14%, 38%, 44% and 46% improvement respectively over the state-of-the-art performance measures. Statistical tests for various Top-K results (i.e., $1 \leq K \leq 10$) also reported significance (i.e., all *p-values*≤ 0.05) of our technique over the state-of-the-art with large effect sizes (i.e., $0.39 \leq \Delta \leq 0.90$).

Our technique outperforms the state-of-the-art approach on relevant API class suggestion, and it suggests relevant API classes with **38**% higher precision and **46**% higher reciprocal rank than those of the state-of-the-art.

D. Evaluation of NLP2API: Query Reformulation

Although our approach outperforms the state-of-the-art on relevant API class suggestion, we further apply the suggested API classes to query reformulations. Then we demonstrate the potential of our reformulated queries for improving the code snippet search. In this section, we also answer RQ_3, RQ_4 and RQ_5 using our experiments as follows:

Answering RQ_3–Improvement of Natural Language Queries with the Suggested API Classes: We reformulate each of the generic natural language queries for code search using the API classes suggested by our technique. Then we investigate the performance of these reformulated queries using code search. We prepare a code corpus of 4,170 code segments where 310 segments are *ground truth* code segments (Section IV-A) and 3,860 code segments were taken from a publicly available and curated dataset [58] based on hundreds of GitHub projects. We normalize these segments using standard natural language preprocessing (i.e., stop and keyword removal, token splitting), and index them with *Lucene*. We then perform code search on this corpus, and contrast between generic natural language queries and our reformulated queries in terms of their Effectiveness and code retrieval performances.

From Table IV, we see that our reformulations improve or preserve 75% (i.e., 48% improvement and 27% preserving) of the given queries. The improvement ratio reaches the maximum of 49% with a reformulation length of 20. According to relevant literature [31, 61, 54], such statistics are promising. Fig. 7 further demonstrates the impact of our reformulations on the baseline generic queries. We see that the baseline natural language queries retrieve ground truth code segments with 50% Top-10 accuracy (dashed line, Fig. 7-(a)) and 0.32 mean reciprocal rank (dashed line, Fig. 7-(b)). On the contrary, our reformulated queries achieve a maximum of 69% Top-10 accuracy with a reciprocal rank of 0.47 which are 37% and 47% higher respectively than the baseline. Quantile analysis in Table V also shows that our provided result ranks are more promising than those of the baseline queries.

Reformulations offered by our technique improve **49**% of the generic natural language queries, and the reformulated queries achieve **37**% higher accuracy and **47**% higher reciprocal rank than those of the generic NL queries.

Answering RQ_4–Comparison with Existing Query Reformulation Techniques: Nie et al. [55] collect pseudo-relevance feedbacks from Stack Overflow on a given query and then apply Rocchio's method to expand the query. Their approach, QECK, outperformed earlier studies [51, 47] on query reformulation targeting code search which made it the state-of-the-art. Another contemporary work, CoCaBu [70] applies VSM in identifying appropriate program elements from Stack Overflow posts. To the best of our knowledge, these are the most recent and most closely related works to ours. Due to the unavailability of authors' prototype, we re-implement them ourselves using their best performing parameters (e.g., PRF size = 5–10, reformulation length = 10), and then compare them with ours. We also compare with RACK [63] in the context of query reformulation due to its related nature.

Table V shows a quantile analysis of the result ranks provided by the existing techniques. If results are returned closer to the top by a reformulated query than its baseline counterpart, we call it *query improved* and vice versa as *query worsened*. We see that CoCaBu and RACK perform relatively higher than QECK. CoCaBu improves 36% and worsens 42% of the 310 baseline queries. On the contrary, our technique improves 48% and worsens 25% of the given queries which are 32% higher and 40% lower respectively than those of CoCaBu. Furthermore, according to the quantile analyses, the extents

TABLE V
COMPARISON OF QUERY EFFECTIVENESS WITH EXISTING QUERY REFORMULATION TECHNIQUES

Technique	#QC	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
QECK [55]	310	72 (23.23%)	139	02	11	74	01	1,861	177 (57.10%)	131	11	35	163	02	1,259	61 (19.68%)
RACK [63]	310	105 (33.87%)	75	02	08	60	01	971	147 (47.42%)	136	07	31	156	02	1,277	58 (18.71%)
CoCaBu [70]	310	**113 (36.45%)**	191	02	14	103	01	2,607	**131 (42.26%)**	102	06	24	91	02	1,567	66 (21.29%)
Baseline	310	-	-	07	25	145	02	1,460	-	-	01	03	15	01	582	-
NLP2API	310	**149 (48.07%)**	**170**	**02**	**12**	**74**	**01**	2,816	**78 (25.16%)**	**75**	**03**	**13**	**59**	**02**	**826**	**83 (26.77%)**
NLP2API$_{max}$	310	**152 (49.03%)**	**172**	**02**	**10**	**61**	**01**	2,926	**69 (22.26%)**	**73**	**03**	**11**	**70**	**02**	**786**	**89 (28.71%)**

Mean = Mean rank of first correct results returned by the queries, $\mathbf{Q}_i$= i^{th} quartile of all ranks considered

TABLE VI
COMPARISON WITH POPULAR WEB/CODE SEARCH ENGINES

Technique	Hit@10	MAP@10	MRR@10
Google	100.00%	64.80%	0.79
NLP2API$_{Google}$	100.00%	**78.13%**	**0.84**
Stack Overflow	91.29%	59.55%	0.67
NLP2API$_{SO}$	91.29%	**80.25%**	**0.87**
GitHub	88.71%	53.46%	0.58
NLP2API$_{GitHub}$	89.03%	**62.26%**	**0.68**

of our rank improvement over the baseline are comparatively higher than the extents of rank worsening which indicates a net benefit of the reformulation operations.

Our technique outperforms the state-of-the-art approaches on query reformulation, and it improves **32**% more and worsens **40**% less queries than those of the state-of-the-art.

Answering RQ$_5$–Comparison with Existing Code/Web Search Engines: Although our approach outperforms the state-of-the-art studies [63, 55] on relevant API suggestion and query reformulation, we further compare with two popular web search engines – *Google, Stack Overflow native search* – and one popular code search engine –*GitHub native search*. Given the enormous and *dynamic index database* and *restrictions* on the *query length* or *type*, a full scale or direct comparison with these search engines is neither feasible nor fair. We thus investigate whether results returned by these contemporary search engines for generic queries could be significantly improved or not with the help of our reformulated queries.

Collection of Search Results and Establishment of Ground Truth: We first collect Top-30 results returned by each search engine for each of the 310 queries. For result collection, we make use of *Google's custom search API* [3] and the native API endpoints provided by Stack Overflow and GitHub. Since our goal is to find relevant code snippets, we adopt a pragmatic approach in the establishment of ground truth for this experiment. In particular, we analyse those 30 results semi-automatically, look for *ground truth code segments* (i.e., collected in Section IV-A) in their contents, and then select Top-10 results as *ground truth search results* that contain either the ground truth code or highly similar code. It should be noted that ground truth code segments and our suggested API classes are taken from two different sources.

Comparison between Initial Search Results and Re-ranked Results with Reformulated Queries: While the search engines return results mostly for the natural language queries, we further re-rank the results with our reformulated queries (i.e., generic search keywords + relevant API classes) using lexical similarity analysis (e.g., cosine similarity [58]). We then evaluate Top-10 results both by each search engine and by our re-ranking approach against the *ground truth search results*, and demonstrate the potential of our reformulations.

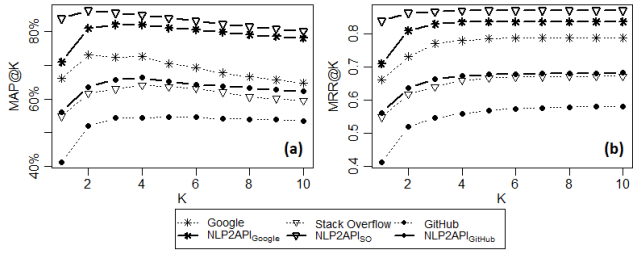

Fig. 8. Comparison between popular web/code search engines and NLP2API in relevant code segment retrieval using (a) MAP@K and (b) MRR@K

From Table VI, we see that our re-ranking approach that leverages our reformulated queries improves the initial search results returned by each of the engines. In particular, the performances are improved in terms of precision and reciprocal rank. For example, Google returns search results with 65% precision and 0.79 reciprocal rank when Top-10 results are considered. Our approach, NLP2API$_{Google}$, improves the ranking and achieves a MAP@10 of 78% and a MRR@10 of 0.84 which are 21% and 6% higher respectively. That is, although Google performs high as a *general purpose* web search engine, it might always not be precise for *code search* due to the lack of appropriate contexts. Our approach incorporates context into the search using relevant API names, and delivers more precise code search results. As shown in Table VI and Fig. 8, similar findings were also achieved against GitHub code search and Stack Overflow native search.

Our technique improves upon the result ranking of all three popular search engines using its reformulated queries, and it achieves $\approx$ **20**% higher precision than Google.

V. THREATS TO VALIDITY

Threats to *internal validity* relate to experimental errors and biases. Re-implementation of the existing techniques could pose a threat. However, we used authors' implementation of RACK [63] and replicated Nie et al. [55] and Sirres et al. [70] carefully. We had multiple runs and found their best performances with the authors' adopted parameters which were finally chosen for comparisons. Thus, threats associated with the re-implementation might be mitigated.

Our code corpus (Section IV-A) contains 4,170 documents including 310 ground truth code segments. It is limited compared to a real life corpus (e.g., GitHub). However, we believe that our corpus might be sufficient enough to *contrast* between a *generic NL query* and a *reformulated query*. Please note that our goal is to *reformulate* a query effectively for code search.

Besides, we compared with three popular search engines and demonstrated the potential of our query reformulations.

Threats to *external validity* relate to generalizability of a technique. Although we experimented with Java based Q & A threads and tasks, our technique can be easily adapted for other programming languages given that code segments and API classes are extracted correctly from Stack Overflow.

VI. Related Work

Relevant API Suggestion: There have been several studies [51, 22, 74, 28, 39, 44] that return relevant functions, API classes and methods against natural language queries. McMillan et al. [51] employ natural language processing (NLP), PageRank and spreading activation network (SAN) on a large corpus (e.g., FreeBSD), and identify functions relevant to a given query. Although they apply advanced approach for function ranking (e.g., PageRank), their candidate functions were selected using simple textual similarity which is subject to vocabulary mismatch issues [26]. On the contrary, we apply pseudo-relevance feedback, PageRank and TF-IDF for selecting the candidate API classes. Chan et al. [22] apply sophisticated graph mining techniques and return relevant API elements as a connected sub-graph. However, mining a large corpus could be very costly. Thung et al. [74] mine API documentations and feature history, and suggest relevant methods for an incoming feature request. However, this approach is project-specific and does not overcome the vocabulary mismatch issues. Rahman et al. [63] apply two heuristics derived from keyword-API co-occurrences in Stack Overflow Q & A threads, and attempt to counteract the vocabulary mismatch issues during API suggestion. Unfortunately, their approach suffers from low precision due to the adoption of simple co-occurrences. On the contrary, we (1) exploit query-API co-occurrence using a skip-gram based probabilistic model (i.e., *fastText* [53, 17]), and (2) employ pseudo-relevance feedback, Borda count and PageRank algorithm, and thus, (3) provide a novel solution that partially overcomes the limitations of earlier approaches. Rahman et al. is the most closely related work to ours in API suggestion. We compare ours with this work, and the detail comparison can be found in Section IV-C. Gvero and Kuncak [28] accept free-form NL queries, perform natural language processing, statistical language modelling on source code and suggest relevant method signatures. There exist other works that provide relevant code for natural language queries [16, 36, 14, 20], test cases [40, 41, 64], structural contexts [34], dependencies [76], and API class types [73, 78]. On the contrary, we collect relevant API classes for free-form NL queries by mining crowd generated knowledge stored in Stack Overflow questions and answers.

Query Reformulation for Code Search: Several earlier studies [32, 39, 28, 75, 47, 44, 43, 55, 50] reformulate a natural language query to improve the search for relevant code or software artefacts. Hill et al. [32] expand a natural language query by collecting frequently co-occurring terms in the method and field signatures. Conversely, we apply a different context (i.e., Q & A pairs) and a more sophisticated co-occurrence mining (e.g., skip-gram model). Lu et al. [47] expand a search query by using part of speech (POS) tagging and WordNet synonyms. Lemos et al. [42] combine WordNet and test cases in the query reformulation. However, WordNet is based on natural language corpora, and existing findings suggest that it might not be effective for synonym suggestion in software contexts [72]. On the contrary, we use a software-specific corpus (e.g., programming Q & A site), and more importantly, apply relevant API classes to query reformulation. Wang et al. [75] employ relevance feedback from developers to improve code search. Recently, Nie et al. [55] collect pseudo-relevance feedback from Stack Overflow, and reformulate a natural language query using Rocchio's method. However, their suggested terms are natural language terms which might not be effective enough for code search given the existing evidence [15]. Another contemporary work [70] simply relies on Lucene to identify appropriate program elements from Stack Overflow answers for query reformulation. On the contrary, we employ PRF, PageRank, TF-IDF, Borda count and extra-large data analytics, and provide relevant API classes for query reformulation. The above two works are the most closely related to ours. We compare with them empirically, and the detail comparison can be found in Section IV-D. There exist other studies that search source code [39, 77], project repository [43], and artefact repository [44] by reformulating natural language queries. There also exist a number of query reformulation techniques [27, 31, 71, 37, 38, 61, 59, 23, 69] for concept/feature/bug/concern location. However, they suggest project-specific terms (e.g., domain terms [29]) rather than relevant API classes (like we do) for query reformulations. Hence, such terms might not be effective enough for code search on a large corpus that contains cross-domain projects.

In short, we meticulously bring together *crowd generated knowledge* [55], *extra-large data analytics* [17], and several IR-based approaches to effectively solve a complex Software Engineering problem, i.e., query reformulation for code search, which was not done by the earlier studies. Our query reformulation technique can also be employed on top of existing code or web search engines for improved code search (i.e., RQ_5).

VII. Conclusion and Future Work

In this paper, we propose a novel technique–NLP2API–that reformulates a natural language query for code search with relevant API classes. We mine Stack Overflow Q & A threads, and employ PageRank, Borda count and large data analytics for identifying the relevant API classes. Experiments with 310 code search queries report that our technique (1) suggests ground truth API classes with 48% precision and 58% recall for 82% of the queries, and (2) improves the given search queries significantly through reformulations. Comparisons with two state-of-the-art techniques and three popular search engines not only validate our empirical findings but also demonstrate the superiority of our technique. In future, we plan to investigate the potential of our skip-gram model based on Stack Overflow for project-specific code search (e.g., concept location, bug localization).

REFERENCES

[1] Borda count. URL https://en.wikipedia.org/wiki/Borda_count.

[2] Codejava. URL http://www.codejava.net.

[3] Google custom search. URL https://developers.google.com/custom-search.

[4] Example code snippet. URL https://goo.gl/BBxPwH.

[5] Backend of github search. URL https://stackoverflow.com/questions/3616221/search-code-inside-a-github-project.

[6] Java2s: Java Tutorials, . URL http://java2s.com.

[7] JavaDB: Java Code Examples, . URL http://www.javadb.com.

[8] Jsoup: Java HTML Parser. URL http://jsoup.org.

[9] KodeJava: Java Examples. URL http://kodejava.org.

[10] Replication package: ICSME 2018. URL https://goo.gl/Meujmx.

[11] How do i decompress a gzip file in java? URL https://goo.gl/14QkXq.

[12] Rack website. URL http://homepage.usask.ca/~masud.rahman/rack.

[13] Stack Exchange archive. URL https://archive.org/download/stackexchange.

[14] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proc. OOPSLA-C*, pages 681–682, 2006.

[15] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *EMSE*, 17(4-5):424–466, 2012.

[16] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. FSE*, pages 157–166, 2010.

[17] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[18] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.

[19] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[20] B. A. Campbell and C. Treude. Nlp2code: Code snippet content assist via natural language tasks. In *Proc. ICSME*, pages 628–632, 2017.

[21] C Carpineto and G Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.*, 44(1):1:1—-1:50, 2012.

[22] W. Chan, H. Cheng, and D. Lo. Searching Connected API Subgraph via Text Phrases. In *Proc. FSE*, pages 10:1–10:11, 2012.

[23] O. Chaparro, J. M. Florez, and A Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proc. ICSME*, page to appear, 2017.

[24] M. Duijn, A. Kucera, and A. Bacchelli. Quality questions need quality code: Classifying code fragments on stack overflow. In *Proc. MSR*, pages 410–413, 2015.

[25] S Ercan, Q Stokkink, and A Bacchelli. Automatic Assessments of Code Explanations: Predicting Answering Times on Stack Overflow. In *Proc. MSR*, pages 442–445, 2015.

[26] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.

[27] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.

[28] T. Gvero and V. Kuncak. Interactive synthesis using free-form queries. In *Proc. ICSE*, pages 689–692, 2015.

[29] S Haiduc and A Marcus. On the Use of Domain Terms in Source Code. In *Proc. ICPC*, pages 113–122, 2008.

[30] S. Haiduc and A. Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, June 2011.

[31] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.

[32] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.

[33] E Hill, S Rao, and A Kak. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Proc. SCAM*, pages 184–193, 2012.

[34] R. Holmes and G.C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*, pages 117–125, 2005.

[35] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[36] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proc. ICSE*, pages 664–675, 2014.

[37] K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.

[38] K. Kevic and T. Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.

[39] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proc. ASE*, pages 376–379, 2011.

[40] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: Using test-cases to search and reuse source code. In *Proc. ASE*, pages 525–526, 2007.

[41] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proc. SAC*, SAC '09, pages 476–482, 2009.

[42] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proc. MSR*, pages 212–221, 2014.

[43] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proc. Internetware*, pages 36–44, 2016.

[44] J. Lin, Y. Liu, J. Guo, J. Cleland-Huang, W. Goss, W. Liu, S. Lohar, N. Monaikul, and A. Rasin. Tiqi: A natural language interface for querying software project data. In *Proc. ASE*, pages 973–977, 2017.

[45] Z. Lin, Y. Zou, J. Zhao, and B. Xie. Improving software text retrieval using conceptual knowledge in source code. In *Proc. ASE*, pages 123–134, 2017.

[46] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. ASE*, pages 234–243, 2007.

[47] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Proc. SANER*, pages 545–549, 2015.

[48] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[49] A Marcus, A Sergeyev, V Rajlich, and J I Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. WCRE*, pages 214–223, 2004.

[50] L. Martie, T. D. LaToza, and A. v. d. Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *Proc. ASE*, pages 24–35, 2015.

[51] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proc. ICSE*, pages 111–120, 2011.

[52] R. Mihalcea and P. Tarau. Textrank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[54] L Moreno, G Bavota, S Haiduc, M Di Penta, R Oliveto, B Russo, and A Marcus. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*, pages 567–578, 2015.

[55] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *TSC*, 9(5):771–783, 2016.

[56] F. J. Ortega, C. Macdonald, J. A. Troyano, and F. Cruz. Spam detection with a content-based random-walk algorithm. In *Proc. SMUC*, pages 45–52, 2010.

[57] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton. Improving Low Quality Stack Overflow Post Detection. In *Proc. ICSME*, pages 541–544, 2014.

[58] M. M. Rahman and C. K. Roy. On the use of context in recommending exception handling code examples. In *Proc. SCAM*, pages 285–294, 2014.

[59] M. M. Rahman and C. K. Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*, pages 220–225, 2016.

[60] M. M. Rahman and C. K. Roy. Improved query reformulation for concept location using coderank and document structures. In *Proc. ASE*, pages 428–439, 2017.

[61] M. M. Rahman and C. K. Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.

[62] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.

[63] M. M. Rahman, C. K. Roy, and D. Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.

[64] Steven P. Reiss. Semantics-based code search. In *Proc. ICSE*, pages 243–253, 2009.

[65] P C Rigby and M P Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.

[66] J.J. Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc.

[67] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *Proc. ESEC/FSE*, pages 191–201, 2015.

[68] G. Salton and C. Buckley. Readings in information retrieval. chapter Improving Retrieval Performance by Relevance Feedback, pages 355–364. 1997.

[69] D Shepherd, Z P Fry, E Hill, L Pollock, and K Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.

[70] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *EMSE*, 2018.

[71] B. Sisman and A. C. Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.

[72] G Sridhara, E Hill, L Pollock, and K Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *Proc. ICPC*, pages 123–132, 2008.

[73] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.

[74] F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proc. ASE*, pages 290–300, 2013.

[75] S. Wang, D. Lo, and L. Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proc. ASE*, pages 677–682, 2014.

[76] F. W. Warr and M. P. Robillard. Suade: Topology-Based Searches for Software Investigation. In *Proc. ICSE*, pages 780–783, 2007.

[77] M. Wrsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proc. ICSE*, pages 165–174, 2010.

[78] T. Xie and J. Pei. MAPO: Mining Api Usages from Open Source Repositories. In *Proc. MSR*, pages 54–57, 2006.

[79] J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.

[80] X Ye, H Shen, X Ma, R Bunescu, and C Liu. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proc. ICSE*, pages 404–415, 2016.

[81] T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.

[82] Y. Zhang, W. Zhang, J. Pei, X. Lin, Q. Lin, and A. Li. Consensus-based ranking of multivalued objects: A generalized borda count approach. *TKDE*, 26(1):83–96, 2014.