Automated Query Reformulation for Efficient Search based on Query Logs From Stack Overflow

Kaibo Cao†, Chunyang Chen††, Sebastian Baltes§, Christoph Treude§, Xiang Chen§§
†Software Institute, Nanjing University, China
‡Faculty of Information Technology, Monash University, Australia
§School of Computer Science, University of Adelaide, Australia
§School of Information Science and Technology, Nantong University, China
imkbcao@gmail.com, chunyang.chen@monash.edu, {sebastian.baltes, christoph.treude}@adelaide.edu.au, xchencs@ntu.edu.cn

Abstract—As a popular Q&A site for programming, Stack Overflow is a treasure for developers. However, the amount of questions and answers on Stack Overflow make it difficult for developers to efficiently locate the information they are looking for. There are two gaps leading to poor search results: the gap between the user’s intention and the textual query, and the semantic gap between the query and the post content. Therefore, developers have to constantly reformulate their queries by correcting misspelled words, adding limitations to certain programming languages or platforms, etc. As query reformulation is tedious for developers, especially for novices, we propose an automated software-specific query reformulation approach based on deep learning. With query logs provided by Stack Overflow, we construct a large-scale query reformulation corpus, including the original queries and corresponding reformulated ones. Our approach trains a Transformer model that can automatically generate candidate reformulated queries when given the user’s original query. The evaluation results show that our approach outperforms five state-of-the-art baselines, and achieves a 5.6% to 33.5% boost in terms of GLEU.

Index Terms—Stack Overflow, Data Mining, Query Reformulation, Deep Learning, Query Logs

I. INTRODUCTION

Stack Overflow is the most popular question and answer (Q&A) site for programming-related knowledge sharing and acquisition. Over the past decade, Stack Overflow has accumulated a large amount of user-generated content, making it a valuable repository of software engineering knowledge. When developers encounter a specific programming question such as how to use this library?, what is the difference between two languages? [1], how to understand this concept? [2], [3], they tend to use Stack Overflow to find answers [4]. To assist developers in finding the knowledge they are looking for in such a large-scale knowledge repository, Stack Overflow provides a search engine [4], which supports free text search as well as an advanced search with metadata filters.

However, even using the provided search engine, it is still not easy for developers to effectively find what they want [5], [6]. There are two reasons for unsatisfactory search results. First, there exists a certain semantic gap between the users’ query intention and input queries. This means that it is difficult for users to accurately express their query intention with a few keywords [7], [8]. For example, a developer wants to search for the usage of nested lists. However, they may not know how to express this concept accurately and may use a term like “list in list” as the query. This query is imprecise, and this kind of semantic gap can pose a significant challenge to the search engine of Stack Overflow. Second, a certain semantic gap exists between the users’ queries and the text content in the relevant posts. It means the same meaning may be described in different ways with few overlapping words. For example, a developer may input the query “sorting in linear time”. However, the relevant post titled “sorting with $O(n)$ complexity” cannot be retrieved by the search engine. Moreover, abbreviations, synonyms, or even misspelling [9]. [10] can also lead to this kind of semantic gap.

To alleviate the above semantic gaps, developers may constantly reformulate their queries until the query reflects their real query intention and leads to relevant posts. By analyzing users’ query logs provided by Stack Exchange Inc., the company behind Stack Overflow, we find that 24.62% of the queries are reformulated before visiting a post, and that developers reformulate their queries 1.46 times on average before clicking on a result. Fig. 1 shows an illustrative example. The user first performs the query “do and while in java”, which yields a large number of irrelevant results. The reason is that the search engine uses or to connect all the words in the query, and the search engine cannot distinguish “do” and “while” as domain-specific keywords. Then, the user adds the word “loop” to this query to make it more explicit, which leads to a potentially desired post. The process of modifying a given query to find a satisfactory search result [11] is called query reformulation. Understanding query reformulation [12]–[16] has become an important issue in designing effective information retrieval systems.

To investigate how developers reformulate their queries, we first perform a formative study on the users’ activity logs from Stack Overflow. Based on the logs from 2,216,219 users between December 2017 and November 2018, we extract 4,631,756 queries from 3,125,427 sessions. After analyzing these queries, we find that query reformulation has certain

* Corresponding Authors

https://stackoverflow.com/search
The evaluation results show that, in terms of \textit{ExactMatch} approach with large-scale archival manual reformulation results. We evaluate the quality of the reformulated queries of our approach \textit{SEQUER} based on an attention-based Transformer. We evaluate the quality of the reformulated queries generated by our approach \textit{SEQUER} with large-scale archival manual reformulation results.

- We implement a browser plugin for supporting automated software-specific query reformulation in practice.

### II. Data Collection

The dataset we used is based on a larger dataset containing all internal HTTP requests processed by Stack Overflow’s web servers within one year (747,421,780 requests from December 2017 to November 2018). Internal means that the dataset only contains requests with a referrer URL from stackoverflow.com. If a user, for example, reached a Stack Overflow post by clicking on a Google search result and then triggered a search within Stack Overflow, only the second (internal) search request would be included in the dataset, not the request for the post having a Google-related referrer. For each HTTP request, the dataset provides an anonymized user identifier that represents logged-in registered users as well as users identified by a browser cookie or users identified by their IP address. This dataset also assigns certain event types to the requests (e.g., searching, post visiting, or question list browsing), depending on their target URL.

Before extracting the event sequences relevant to this study, we preprocess the data as follows. First, we group all events per user identifier and then order them chronologically. Second, we utilize heuristics based on the timestamps and request targets to filter out bot traffic and event sequences merely consisting of page refreshes. Third, to distinguish between individual sessions, we group the events into sequences of events that are not more than six minutes apart, following Sadowski et al.’s approach [20]. Finally, we add an additional filtering step to avoid gaps in the data caused by the focus on internal requests. A user may, for example, follow external links in Stack Overflow posts and then navigate back to Stack Overflow or open multiple posts in parallel browser tabs. For our analysis of query reformulation on Stack Overflow, we focus on complete linear navigation sequence, that is sequence where the referer of one request matches the target URL of the previous request.

After the above data preprocessing, we get a dataset of complete linear navigation sequences. As shown in the table in Fig. 1 each event is represented by one row with five attributes: \textit{RootEventId}, \textit{EventId}, \textit{EventTime}, \textit{EventType}, and \textit{URL}.
and URL. Specifically, the `RootEventId` refers to the `EventId` of the first event in the session, the `EventId` is a unique ID that can identify an event, the `EventTime` indicates the UTC time when the event occurred, the `EventType` shows the type of the event (possible values of event type are `{Search,Post,QuestionsList,Home,Tags,PostHistory}`), the `URL` is the web request that triggers this event, which contains, for example, the query content or the post ID, depending on the `EventType`.

Our dataset contains 42,173,522 events from 16,164,506 sessions generated by 9,712,878 users, in which 9,046,179 events are queries. On Stack Overflow, the post visit event and the query event are the two most common operations performed by the users, accounting for 46.21% and 21.45% of the events respectively. Users often reach a post in three ways: links in the post (58.75%), search (17.25%), and question list (5.69%). That is, in addition to the navigation between posts, search is the most common way for the users to find a post.

From the session perspective, 30.82% of the sessions contain query event(s), and the users perform an average of 1.82 queries in these sessions. Fig. 2 shows the distribution of the number of sessions and their average session duration in terms of the number of queries contained in the session. In this figure, we can find as the number of queries in the session increases, the number of such sessions decreases exponentially, while the session duration increases linearly.

Fig. 2: The distribution of sessions number and duration in terms of the number of queries contained in the session.

To train our query reformulation model, we obtain the query reformulation records (i.e., the user’s process of transforming an original query into a better one) from the user’s navigation sequence. We call these processes the query reformulation threads. We first remove the sessions that do not contain a query event or of which the last event is not a post visit. The latter limitation is to ensure that the users finish the query reformulation with a desired result. This yields a dataset containing 8,546,915 events from 3,125,427 sessions generated by 2,216,219 users, in which 4,631,756 events are queries. We adopt a pattern-based method to extract the query reformulation threads from each session following a greedy approach:

\[ q_1, q_2, \cdots, q_i, p_1 (optional), q_{i+1}, \cdots, q_m, p_m, \cdots \]

In this pattern, we use \( q_i \) to denote the \( i \)-th query event and use \( p_i \) to denote the \( i \)-th post visit event. All the events are ordered chronologically. We conjecture that \( q_n \) is a relatively better query compared to the queries from \( q_1 \) to \( q_{n-1} \). Sometimes users may need to visit posts in the query results to determine whether a particular result is what they want, and they may reformulate their queries again after visiting those posts to get better results. Therefore, to make the pattern more versatile and avoid misrecognition of reformulated queries, post visit event is allowed in the sequence of query events. More details on extracting query reformulation threads can be found in Section IV-B.

III. EMPIRICAL STUDY OF QUERY REFORMULATION ON STACK OVERFLOW

In this section, to motivate the required tool support, we perform a formative study to understand the characteristics of query reformulation by analyzing the Stack Overflow log data.

A. What are the characteristics of queries?

Query content analysis. We analyze the query strings to investigate what the users are searching for on Stack Overflow. First, we collect all query strings and apply traditional text processing steps (i.e., removing punctuation, transforming to lower case, excluding stop words) to them. Then, we identify the most popular \( n \)-grams in the queries.

Table I shows the top-10 most frequent 1-grams, 2-grams, 3-grams, and 4-grams in the users’ queries. Programming languages such as `Python` and `Java`, platforms such as `Android`, data types such as `string`, and data structures such as `array` are the most frequently queried terms. At the same time, “\( \text{how to} \)” is the most frequently used phrase in the queries. Almost every 3-gram starts with “\( \text{how to} \)”, next comes “\( \text{what is} \)” and programming language qualifiers such as “\( \text{in python} \)” and “\( \text{in java} \)”. It is worth noting that some `Java` and `Python` error logs appear in the top ten 4-grams. The reason is that the developers often paste these error logs directly into the search box to perform queries, and logs like “exception in thread "main"” and “ImportError: No module named” are the most common error types.

<table>
<thead>
<tr>
<th>Rank</th>
<th>1-gram</th>
<th>2-gram</th>
<th>3-gram</th>
<th>4-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>python</td>
<td>how to</td>
<td>how to create</td>
<td>how to create</td>
</tr>
<tr>
<td>2</td>
<td>java</td>
<td>in python</td>
<td>how to get</td>
<td>how to make</td>
</tr>
<tr>
<td>3</td>
<td>file</td>
<td>what is</td>
<td>how do i</td>
<td>how do i use</td>
</tr>
<tr>
<td>4</td>
<td>string</td>
<td>in java</td>
<td>how to create</td>
<td>exception in thread &quot;main&quot;</td>
</tr>
<tr>
<td>5</td>
<td>android</td>
<td>failed to</td>
<td>how to make</td>
<td>ImportError: No module named</td>
</tr>
<tr>
<td>6</td>
<td>c#</td>
<td>in swift</td>
<td>could not find</td>
<td>how to get the</td>
</tr>
<tr>
<td>7</td>
<td>error</td>
<td>in r</td>
<td>how to change</td>
<td>no such file or</td>
</tr>
<tr>
<td>8</td>
<td>array</td>
<td>unable to</td>
<td>how can i</td>
<td>how to add a</td>
</tr>
<tr>
<td>9</td>
<td>sql</td>
<td>how do</td>
<td>how to find</td>
<td>how to check if</td>
</tr>
<tr>
<td>10</td>
<td>list</td>
<td>not found</td>
<td>how to install</td>
<td>such file or directory</td>
</tr>
</tbody>
</table>

Query length analysis. The users may intentionally limit a query’s length to avoid returning a few or even empty results when using traditional search engines [21]. To investigate whether this phenomenon exists when the users perform queries on Stack Overflow, we use whitespaces as the separator to compute the query length (i.e., word count). Note that we treat words synthesized by CamelCase or underscore_case as
one word since these words often appear in code snippets and can be regarded as the identifiers of variables, classes, or methods.

Fig. 3 shows the distribution of query length via a box plot. The median, mean, 25th percentile, and 75th percentile of the length are 3, 3.6, 2, and 4 respectively. The distribution shows that users do intentionally limit the length of their queries to get better search results. However, we can also easily find that the query length span of the outliers is very large, and all values from 8 to 25 correspond to outliers. For these queries, it is difficult for the search engine of Stack Overflow to return satisfactory results. Because most search engines are optimized only to handle common requests, they use exact-match techniques in which all query words must match a web page for web page retrieval [22]. A longer query means lower matching probability and leads to lower-quality search results. After manual analysis, we find that most of these queries contain error messages or code snippets.

**Advanced search methods analysis.** We notice that some of the queries in the log are structured, which means the users use advanced search methods [23]. Advanced search methods can provide users with a convenient way to filter search results. For example, the users can narrow the search results by only considering the posts with the tag python or the posts that have a minimum score of 500. Stack Overflow provides 29 advanced search methods to help the users filter search results. We first apply regular expressions to the users’ queries to identify the used advanced search method(s). Then we calculate the ratio of the queries using advanced search and the proportion of each advanced search method.

The result shows that 8.74% of the queries use the advanced search. The top-10 most frequently used advanced search methods, an illustrative example, and their proportions can be found in Table II. Tag filtering, user filtering, and declaring specific phrases are the top-3 most commonly-used advanced search methods, accounting for 73.53%, 10.35%, and 10.00% respectively.

**B. Why are queries reformulated?**

The users reformulate their queries for many reasons. For example, the queries may not specify the corresponding programming language, or some words in the queries may be misspelled. We use the provided dataset to analyze why the users reformulate their queries. We manually classify 384 randomly collected query reformulation threads [4] into four categories. The first and second author classified these 384 threads independently. For the cases without an agreement, the final classification result is determined through discussions. For the threads with more than two queries, if multiple query reformulation types are identified, the type of reformulation from the first query to the last query is used as the final classification result. The Kappa inter-rater agreement [24] is 0.83, which shows the high agreement of classification.

Table III shows the manual classification results. In this table, we can see that adding new information to the query is the most common query reformulation operation, which accounts for 40.1% of all threads, then comes modifying the query (33.59%), and deleting information from the query (23.18%). In the category of adding new information [25], we further divide it into two sub-categories: (1) adding information about specific programming languages or platforms, (2) adding new requirements for the question or more detailed content. In the category of modifying, we further divide it into three sub-categories: (1) spelling and syntax checking [26], (2) simplifying and refining the query to reformulate it into the most commonly-used expression, (3) modifying to other content related to the original query. In the category of deleting, we further divide it into three sub-categories: (1) deleting some unnecessary or less informative words, (2) deleting specific information in error messages or code snippets (such as file path, URL, function names, etc.), (3) deleting punctuation and some mistyped symbols.

Based on the classification result, we can observe many different types of query reformulation operations (such as adding information, deleting information, and modifying their expressions). Previous studies [27–30] mainly use rule-based methods to perform automated query reformulation. However, each of these methods can only target query reformulation for a specific reason. Different rules need to be designed for each situation and then implemented with different methods, which is inefficient and difficult to achieve. For example, for the query reformulation in the category of modifying,

![Fig. 3: The distribution of query length via a box plot.](https://stackoverflow.com/help/searching)
the user may completely change the expression of a query (e.g., from “how to cut youtube embedded videos” to “how to make embedded videos that only play certain parts”). It is challenging to implement this type of reformulation by using rule-based methods. Therefore, we find it necessary to propose a general query reformulation approach, which is the motivation of this study.

C. What is the scale of changes that query reformulation involves?

In Section III-B, we find that when reformulating a query, the users may add, remove, or replace some words in the original query. To understand the scale of changes that query reformulation involves, we measure the similarity between the original query and the reformulated query. For each query reformulation thread, \( n - 1 \) pairs of reformulation samples \((original, reformulated)\) can be extracted, that is \( \{(q_1, q_n), (q_2, q_n), \ldots, (q_{n-1}, q_n)\} \). The first query \( q_1 \) in the pair is the initial query performed by the user, and the second query \( q_n \) is the user’s manual reformulated query, which meets their requirement. We use a text-matching algorithm with improved dynamic programming \([31]\) to find the character-level Longest Common Subsequence (LCS) between \( original \) and \( reformulated \). Then the similarity between \( original \) and \( reformulated \) can be defined as follows:

\[
similarity(original, reformulated) = \frac{2 \times N_{match}}{N_{total}} \tag{1}
\]

where \( N_{match} \) is the number of characters in the LCS, and \( N_{total} \) is the sum of the number of characters in \( original \) and \( reformulated \). The similarity score is in the range of 0 to 1. The higher the similarity score, the fewer changes between \( original \) and \( reformulated \).

As shown in Fig. 4, among 1,121,185 query reformulation pairs, 58.07% of the pairs are very similar (i.e., the similarity score is larger than 0.7) with an average of 6.73 character level modifications, which corresponds to 1.14 words according to the average length of query words. The analysis result indicates that most of the query reformulations only involve minor changes.

D. Summary and implications

Our empirical study shows that: (1) On Stack Overflow, most of the users’ queries are simple and short with only 2 to 4 words, but there are also very long queries containing error logs, code snippets, and structured queries with advanced search patterns. (2) The users perform query reformulation for various reasons, making it difficult to design a general query reformulation model by only using rule-based methods. (3) Most query reformulations only involve minor changes.

Considering the diversity of query reformulation patterns, it would require significant effort to manually develop a complete set of reformulation patterns, which is both time-consuming and error-prone. Unlike enumerating all the rules, the deep learning method \([32]\) can automatically learn the latent features from the dataset and these latent features are helpful for downstream tasks.

Based on the above findings, we believe it is necessary and feasible to propose an automated query reformulation approach based on deep learning. Our proposed approach would benefit the developers and the Stack Overflow community. In particular, the developers can use our automated query reformulation tool to refine their queries and get better search results, which can alleviate the effort of manual reformulation. The Stack Overflow community can also use the query reformulation model to enhance the users’ search experience.
IV. AUTOMATED QUERY REFORMULATION

A. Overview of SEQUER

Our formative study has shown a wide variety of query reformulation behaviors. Since not all reformulation patterns are amenable to automated tool support, we focus on query reformulations that trigger minor changes to the original query to develop our automated query reformulation approach. In this work, we formulate query reformulation as a machine translation problem, in which an original query is “translated” into a reformulated query. We solve the problem using the Transformer model [33].

The overall workflow of SEQUER is shown in Fig. 5. SEQUER first extracts query reformulation threads for sampling original queries and corresponding reformulated ones. Then a large corpus of query reformulation pairs is gathered for training the model, each of which contains the original query and the corresponding reformulated one that triggers only minor changes (in Section IV-B). To model the patterns of query reformulation (such as spelling correction, expression refinement, unnecessary word deletion), SEQUER trains a Transformer-based model with a large parallel corpus of query reformulation pairs (from Section IV-C to Section IV-E). Given the users’ original queries, the trained model can suggest a list of reformulated candidates for selection.

B. Collection of query reformulation pairs

Based on the description in Section II, query reformulation threads can be extracted from the users’ navigation sequences by pattern matching. The most important thing for the pattern is how to ensure that \( q_i \) and \( q_{i+1} \) are issued for the same purpose, rather than two independent queries. Two constraints are applied: (1) The character-level similarity between \( q_i \) and \( q_{i+1} \) must be greater than 0.7 by using Equation (1). (2) The browsing time of the post \( p_i \) cannot be greater than 30 seconds. The first constraint can limit the change scale since a large-scale change is likely to introduce a new query. The second constraint can guarantee the post \( p_i \) is not what the user wants, since a previous study [34] shows for an unsatisfactory post, the user would not spend more than 30 seconds on it.

We set the minimum number of query events before the final post visiting event in the thread to 2. After identifying all the query reformulation threads, we perform the following data cleaning steps. For consecutive identical queries in the thread, we only keep one of these identical queries and remove the rest to avoid generating duplicate query reformulation pairs. We also remove threads containing queries with non-English words, since non-English words are not encouraged to be used on Stack Overflow. After the above data cleaning, we pair each query between \( q_1 \) to \( q_{n-1} \) with \( q_n \) (i.e., \( (q_1, q_n), (q_2, q_n), \cdots, (q_{n-1}, q_n) \)) as the training instances. From the provided dataset, we extract a total of 1,121,185 query reformulation pairs.

Some of these reformulation pairs are hard-to-predict due to their large-scale modifications. After randomly sampling reformulation pairs with different similarity, we carry out a pilot study to determine the similarity threshold, which can be used to identify large-scale modifications. The results show that reformulation pairs with a similarity score higher than 0.7 are predictable, while the remaining pairs (i.e., reformulating by large-scale modifications) cannot be predicted even by humans. As analyzed in Section III-C, 58.06% of the query reformulations involve minor modifications with a similarity of at least 0.7. Therefore, we only consider these similar-enough query reformulations, which results in 651,036 query reformulation pairs.

C. Byte pair encoding

Traditional word segmentation methods include whitespace-separated word level and character level segmentation. However, word-level embedding cannot handle the Out-Of-Vocabulary (OOV) problem [35], and the character level is too fine-grained, resulting in the loss of high-level information. Therefore, we adopt the BPE (Byte Pair Encoding) [36], which can effectively interpolate between word-level inputs for frequent symbol sequences and character-level inputs for infrequent symbol sequences.

The BPE algorithm contains multiple iterations. In each iteration, it calculates each consecutive byte pair’s frequency and finds the most frequent one, then merges the two byte pair tokens into one token. This encoding approach can divide words into sub-words like encoding the common words at the word level while encoding the rare words at the character level. Since the form of the users’ queries on Stack Overflow varies, using BPE can better identify the content of the users’ queries. For example, misspelled words can be divided into several correctly spelled sub-words to alleviate the OOV problem’s impact. Words with camel-case style can be separated into several sub-words and then each sub-word can be identified. Besides, using BPE to separate words with their affixes can help the model learn the relationships between them. After applying the BPE algorithm on the corpus, we get a vocabulary composed of sub-words, which is used as the dictionary for our model.

D. Transformer

With the introduction of the attention mechanism [37], many neural machine translation approaches integrate the attention mechanism with sequence transduction models like Convolution Neural Network (CNN) and Recurrent Neural Network (RNN) to improve their performance. Even so, the CNN-based network architectures require many layers to capture long-term dependencies, leading to high computational cost, and operations in RNN-based network structures cannot be parallelized, resulting in low efficiency. To address these issues, Transformer [33] is proposed, which is the first transduction model entirely relying on the attention mechanism and has shown competitive performance on various tasks (such as machine translation [33], image captioning [38], document generation [39], and syntactic parsing [40]).
The most significant difference between Transformer and other sequence transduction models like CNNs and RNNs is that it relies entirely on self-attention (called “Scaled Dot-Product Attention”) to obtain global dependencies, and the attention weights are defined by how each word of the sequence is influenced by all the other words in the sequence. The self-attention mechanism creates shortcuts between the current token and all the other context tokens to determine the current token vector for the final input representation. As weights of these shortcuts are customizable, the self-attention mechanism is able to capture global dependencies without using many layers of convolution and pooling in CNN-based models. At the same time, the calculation in self-attention is implemented with highly optimized matrix multiplication, and thus resolves the low efficiency caused by the RNN-based models, which sequentially encode the input tokens. Intuitively, for an input sequence, first, a neural network is employed to map the input into three matrices: query $Q$, key $K$, and value $V$. Then, the dot products of the queries $Q$ with all keys $K$ is divided by $\sqrt{d_k}$ ($d_k$ is the dimension of the queries), and a softmax function is applied to obtain the weights for the values. Finally, the weighted value is used as the representation of each input.

$$\text{Attention} (Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V \tag{2}$$

Instead of performing a single self-attention function, the Transformer employs a multi-head self-attention, which linearly projects the queries $Q$, keys $K$ and values $V$ $h$ times with different, learned linear projections respectively, where $h$ is the number of heads. To obtain the temporal relationship of the words, the Transformer adds positional embedding to the input embedding.

Self-attention can be regarded as a basic calculation in Transformer. The Transformer model comprises an encoder and a decoder, which are actually multiple identical encoder and decoder blocks stacked on top of each other with the same number of units. Each encoder block has one layer of multi-head self-attention followed by another layer of Feed Forward Network (FFN). On the other hand, each decoder has an extra masked multi-head self-attention, which prevents the model from seeing the generated words during parallel training. On the encoder side, the multi-head self-attention layer’s input is the input embedding with temporal information, and the layer output is normalized and sent into an FFN, which consists of two linear transformations with a ReLU activation. The output of the encoder on the top of the stack is a set of attention vectors $K$ and $V$, which are used by the decoder to determine the token it should pay attention to. On the decoder side, the previous output is used as the input to the masked multi-head self-attention layer. After that, another multi-head self-attention layer with subsequent FFN generates decoder output $h_t$ by getting the query matrix $Q$ from the masked multi-head self-attention layer, the key $K$ and value $V$ matrices from the output of the encoder stack. Finally, the output of the decoder stack $h_t$ is sent to a fully connected neural network to get the logits vectors, and then a softmax layer to predict the probabilities of the next token.

$$P(w_{t+1}|w_1, \cdots, w_t) = \text{softmax}(h_tW + b) \tag{3}$$

where $h_t$ is the output of the decoder stack.

**E. Beam search**

During decoding, for each time step $t$, the Transformer model will output the word with the highest conditional probability $y_t = \arg\max_{y \in \mathcal{D}} P(y|y_1, \cdots, y_{t-1})$ via greedy search from $|\mathcal{D}|$ number of words, where $\mathcal{D}$ represents all the words in the word dictionary. Since we calculate the conditional probability of generating an output sequence based on the input sequence $\prod_{t=1}^T P(y_t|y_1, \cdots, y_{t-1})$, where $T$ is the maximum length of the output sequence, the main problem with greedy search is that there is no guarantee that the optimal sequence will be obtained. The reason is that although the greedy strategy ensures that the output candidate with the highest probability is picked up for each time step, it cannot ensure that the conditional probability of the entire output sequence obtained is the highest.

Beam search [41] is an improved algorithm of greedy search, and beam size $k$ is its hyper-parameter. The decoding process using beam search is as follows: At time step 1, $k$ words with the highest probability are selected as the first word of $k$ candidate output sequences $c_{S1}$. Then, at time step $i$, $k$ output sequences with the highest conditional probability $c_{Si}$ will be selected from $k|\mathcal{D}|$ possible output sequences based on $c_{S1-1}$. After $T$ iterations, $k$ sequences with the highest score will be selected from $CS$ as the beam search result, where $T$ is the maximum number of tokens of the output sequence, and $CS$ is the collection from $c_{S1}$ to $c_{Si}$. Note, for each sequence, portions including and after special end-of-sequence tokens are discarded. The score is calculated as follows:
The hyper-parameter k GPU (32G memory) for about 8 hours. During decoding, learning_rate is set to 0 to ensure the error rate. We train our model with batch_size back propagation \[43\] with Adam optimizer \[44\] to minimize the probability of finding the optimal solution. The length normalization parameter \(\alpha\) is set to 0.6, which is a common practice in neural machine translation \[45\].

To make our work more practical, we develop a browser plugin based on Tampermonkey, which is a popular user-script manager. The browser plugin will automatically analyze the query content and recommend the top-10 query reformulation candidates to the users for selection (a screenshot can be found in Fig. 6). Although the plugin is designed to only work on Stack Overflow now, it can be easily extended to other software-specific Q&A sites.

Fig. 6: A screenshot of our query reformulation plugin for the search engine of Stack Overflow

\[
\frac{1}{L^\alpha} \log P (y_1, \cdots, y_L) = \frac{1}{L^\alpha} \sum_{t=1}^{L} \log P (y_t | y_1, \cdots, y_{t-1}) \tag{4}
\]

where \(L\) is the length of the sequence in \(CS\) and \(\alpha\) is the length normalization parameter.

F. Implementation

In our implementation, the maximum vocabulary size for BPE \[36\] is set to 10,000. For the Transformer, we use the tensor2tensor library \[42\] developed by the Google Brain team. The Transformer model contains four attention heads with four encoder and decoder layers, with hidden_size = 512. During the model training, the parameters are learned by back propagation \[43\] with Adam optimizer \[44\] to minimize the error rate. We train our model with batch_size = 256, learning_rate = 0.0001 for 147 epochs on 4 Nvidia V100 GPU (32G memory) for about 8 hours. During decoding, the hyper-parameter \(k\) of beam search is set to 10 to ensure the probability of finding the optimal solution. The length normalization parameter \(\alpha\) is set to 0.6, which is a common practice in neural machine translation \[45\].

To make our work more practical, we develop a browser plugin based on Tampermonkey, which is a popular user-script manager. The browser plugin will automatically analyze the query content and recommend the top-10 query reformulation candidates to the users for selection (a screenshot can be found in Fig. 6). Although the plugin is designed to only work on Stack Overflow now, it can be easily extended to other software-specific Q&A sites.

V. QUALITY OF RECOMMENDED QUERY REFORMULATION

A. Dataset

From the 651,036 query reformulation pairs, we randomly take 520,830 (80%) of these query pairs as the training set to train the model, 65,103 (10%) as the validation set to tune model hyper-parameters, and 65,103 (10%) as the testing set to evaluate the quality of recommended reformulations.

https://github.com/kbcao/sequer

B. Baselines

As analyzed in Section III-B, many queries are reformulated to fix grammatical errors. Therefore, we first adopt a popular grammatical error correction (GEC) tool as the baseline. LanguageTool\[37\] is an open-source proof-reading tool for more than 20 languages. This tool’s style and grammar checker is rule-based and has been developed for over ten years.

As our task is query reformulation, and Google is the most popular search engine in the world, we choose Google Prediction Service (GooglePS) \[19\] as a baseline. Google uses a prediction service to help complete searches in the search box or address bar within Chrome. These suggestions are based on the real searches that happen on Google. Common and trending queries relevant to the strings entered by the users are shown in the drop-down bar for the users to choose. Since Google has accumulated a large-scale dataset of the users’ queries, GooglePS can efficiently and accurately reformulate users’ queries, such as correcting misspelled words, completing words that the users are typing, and appending the next possible word.

The query reformulation task can also be regarded as a translation task (i.e., translating the original query into the reformulated query). Therefore, we take the most classical neural machine translation model seq2seq \[17\] as our baseline. It contains one Long Short-Term Memory (LSTM) model \[46\] as the encoder, which can encode the original query to an embedding vector, and another LSTM model, which can decode that embedding vector to the reformulated query. We also add the seq2seq model with the attention mechanism as another baseline.

Besides, there are many studies in information retrieval \[47\]–\[50\] about query suggestion. We select HRED-qs \[18\] as one of our baselines, which is a representative hierarchical and session-based query suggestion model with full source code release. HRED-qs is trained with our dataset. In particular, given a query in the session, HRED-qs first encodes the information seen up to the position by a query-level RNN encoder and a session-level RNN encoder. Then it uses the following decoder to predict the next query. To keep the setting of all baselines consistent, we only feed the last query before the reformulated one in the session to HRED-qs.

To make a fair comparison, we employ the same data preprocessing method and Byte Pair Encoding for seq2seq and HRED-qs as SEQUER, and we perform hyper-parameter optimization with grid search \[51\].

C. Evaluation metrics

Query reformulation is similar to the grammar error correction task (i.e., revising some words in the original sentence for generating the target sentence). Therefore, we evaluate the query reformulation with the metrics used in the GEC task (i.e., GLEU, M^2 and ExactMatch).

https://languagetool.org

https://www.blog.google/products/search/how-google-autocomplete-works-search
**GLEU** (General Language Evaluation Understanding) [52], [53] is a customized metric from BLEU (BiLingual Evaluation Understudy) [54], which is a widely used metric to measure the performance of machine translation approaches. Since only part of the source sentence will be changed in the GEC task, which is different from the machine translation task, this motivates a small change to BLEU that computes n-gram precision over the reference but assigns more weight to n-grams that have been correctly changed from the source. Therefore, compared with BLEU, GLEU is more suitable for evaluating query reformulation in our study.

MaxMatch (or M\(^2\)) [55] is another widely used GEC evaluation metric that computes the sequence of phrase-level edits between a source sentence and a system hypothesis that achieves the highest overlap with the gold-standard annotation. These edits are scored by precision, recall, and F\(_1\). Specifically, in the scenario of query reformulation, M\(^2\)@P represents the proportion of edits of the original query given by an approach that appears in the user’s manual reformulation. M\(^2\)@R represents the proportion of edits of the original query by users that are correctly predicted by an approach. M\(^2\)@F\(_1\) is the harmonic mean of M\(^2\)@P and M\(^2\)@R.

ExactMatch (EM) evaluates the probability of a perfect match between the query provided by a specific approach and the user’s manually reformulated one. Since SEQUER uses beam search during decoding, it can suggest multiple reformulations for an original query, for the decoding results of SEQUER with different beam size, we can calculate EM@1, EM@5, and EM@10, where EM@n means one case will be considered positive as long as one of n reformulation results returned by beam search matches the ground truth.

### D. Evaluation results

We report our evaluation results by answering the following two research questions.

1) **RQ1**: Can our approach SEQUER generate better reformulated queries than the baselines?

Table IV shows the evaluation results of query reformulation on the testing set of SEQUER and baselines in terms of all the evaluation metrics. SEQUER outperforms all state-of-the-art baselines by significant margins in terms of all metrics. Compared with the best baseline seq2seq (with attention), SEQUER achieves 5.6% and 4.75% improvement in terms of EM@10 and GLEU. To better illustrate our results, Table IV lists examples of query reformulations by different approaches, and more examples can be found on our project site[7].

GEC tools perform well in correcting misspellings ("strin" to "string" in Example 1), grammar issues ("playing" to "play" in Example 2), and sentence format ("current date" to "current date" in Example 3). However, they only achieve a 5.87% exact match, as spelling errors only account for 19% of the reasons for the users’ query reformulations (in Section III-B). In addition, they have difficulty in detecting spelling errors specific to the programming field. For example, they cannot detect any misspelled words in "how to import bumpy array" where the word "bumpy" is the wrong spelling of "numpy" (a Python library). Instead, SEQUER can correct these software-specific misspelled words by learning the domain knowledge from our software-specific dataset (another example is from "C3" to "C#" in Example 4).

GooglePS can perform complex reformulations by learning from billions of queries Google processes every day, such as adding important missing keywords ("loop" in Example 5) and suggesting language/platform limitations ("python" in Examples 1 and 6). However, as a general-purpose search engine, Google does not perform well in software-specific query reformulation, especially for unpopular software-specific queries. For example, GooglePS cannot reformulate the query "usr/bin/ld: skipping incompatible libpthread.so" in Example 7, but SEQUER can remove the file directory to make it more general. As the file directory often varies between developers’ coding environments, it should be removed to keep the query more general, for retrieving more accurate results. By learning from massive query reformulations, which are software-specific from Stack Overflow, SEQUER can effectively revise such queries by applying software-specific reformulation strategies (similar examples in Examples 8 and 9).

Since using the same training data as SEQUER, the HRED-qs, seq2seq, and seq2seq with attention model can capture software-specific semantics during query reformulation. This is the reason why they achieve better performance than the other baselines. However, since the goal of HRED-qs is basically "next query prediction", the context-aware hierarchical encoding does not enhance the ability of the decoder to generate a reformulated query, but may obscure the information of the original query, causing a deviation of the decoder result. For example, "combo box lost focus" should be reformulated to "combo box lost focus", but HRED-qs reformulates it to "iphone x lost focus". Besides, the problem for both HRED-qs and seq2seq is that the input is encoded into one single vector representation, which may not be sufficient to store all the information, especially for long queries. For example, "Allow user to paste url with .php and return clean url" should be reformulated to "Allow user to paste url with .php and return clean url". However, seq2seq model can only reformulate it to "Allow user to paste url with .php" due to the query length. On the contrary, SEQUER does not have this problem by using an attention-based Transformer model.

In addition to the types of query reformulations mentioned above, SEQUER also outperforms baselines in more complex reformulations such as revising with more commonly-used

---

**TABLE IV: Performance of automated query reformulation**

<table>
<thead>
<tr>
<th>Approach</th>
<th>EM@1</th>
<th>EM@5</th>
<th>EM@10</th>
<th>GLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LanguageTool</td>
<td>—</td>
<td>—</td>
<td>5.87</td>
<td>53.27</td>
</tr>
<tr>
<td>GooglePS</td>
<td>6.20</td>
<td>8.14</td>
<td>8.58</td>
<td>61.07</td>
</tr>
<tr>
<td>HRED-qs</td>
<td>10.40</td>
<td>17.63</td>
<td>20.01</td>
<td>56.85</td>
</tr>
<tr>
<td>seq2seq</td>
<td>11.02</td>
<td>23.11</td>
<td>28.23</td>
<td>61.30</td>
</tr>
<tr>
<td>seq2seq+Attn.</td>
<td>14.53</td>
<td>28.47</td>
<td>33.77</td>
<td>62.93</td>
</tr>
<tr>
<td>SEQUER</td>
<td>22.21</td>
<td>33.47</td>
<td>39.37</td>
<td>67.68</td>
</tr>
</tbody>
</table>

TABLE V: Examples of query reformulation by different approaches (“—” represents no reformulation suggestions)

<table>
<thead>
<tr>
<th>ID</th>
<th>Original Query</th>
<th>GooglePS</th>
<th>seq2seq + Attn.</th>
<th>SEQUER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pandas delete last characters in string</td>
<td>python delete last characters in string</td>
<td>pandas delete last characters in string</td>
<td>python delete last characters in string</td>
</tr>
<tr>
<td>2</td>
<td>playing sound in swift 3</td>
<td>—</td>
<td>—</td>
<td>play sound in swift 3</td>
</tr>
<tr>
<td>3</td>
<td>swift grab currentdate</td>
<td>—</td>
<td>swift grab currentdate</td>
<td>swift get current date</td>
</tr>
<tr>
<td>4</td>
<td>assign string to number C3</td>
<td>—</td>
<td>assign string to number C#</td>
<td>assign string to number C#</td>
</tr>
<tr>
<td>5</td>
<td>do and while in java</td>
<td>do and while loop in java</td>
<td>do and while loop in java</td>
<td>do and while loop in java</td>
</tr>
<tr>
<td>6</td>
<td>requests negotiate</td>
<td>requests negotiate python</td>
<td>python requests negotiate</td>
<td>python requests negotiate</td>
</tr>
<tr>
<td>7</td>
<td>/usr/bin/ld: skipping incompatible libpthread.so</td>
<td>—</td>
<td>libpthreads skipping incompatible libpthread.so</td>
<td>libpthreads skipping incompatible libpthread.so</td>
</tr>
<tr>
<td>8</td>
<td>subtracting the pandas series df from all columns</td>
<td>python subtracting the pandas series df from all columns</td>
<td>subtracting the pandas series df from all columns</td>
<td>subtracting the pandas series df from all columns</td>
</tr>
<tr>
<td>10</td>
<td>opencv scale image</td>
<td>opencv scale image to 0 1</td>
<td>opencv scale image to 0 1</td>
<td>opencv scale image to 0 1</td>
</tr>
<tr>
<td>11</td>
<td>a* search</td>
<td>a* search example</td>
<td>a* search python</td>
<td>a* search python</td>
</tr>
<tr>
<td>12</td>
<td>df -h show disk</td>
<td>—</td>
<td>df -h show disk</td>
<td>df -h show disk</td>
</tr>
<tr>
<td>13</td>
<td>resteasy how to support websocket</td>
<td>—</td>
<td>resteasy how to support websocket</td>
<td>resteasy how to support websocket</td>
</tr>
<tr>
<td>14</td>
<td>volume control programmatically android</td>
<td>android volume control programmatically</td>
<td>android volume control programmatically</td>
<td>android volume control android</td>
</tr>
</tbody>
</table>

Software-specific terms (e.g., “swift grab currentdate” to “swift get current date” in Example 3, “opencv scale image” to “opencv resize image” in Example 10), replace symbols with text (e.g., “a* search” to “a star search” in Example 11) or enclose symbols that will accidentally trigger advanced search in quotation marks (e.g., “df h show disk” to “df -h show disk” in Example 12 to prevent showing results for “df show disk” and not containing “H”), and simplify or refine the query (e.g., “resteasy how to support websocket” to “resteasy websocket” in Example 13 and “volume control programmatically android” to “volume control android” in Example 14).

2) RQ2: What types of query reformulations are challenging for SEQUER to deal with?:

Although SEQUER can achieve the best performance compared to state-of-the-art baselines, SEQUER also makes mistakes in some query reformulations. We manually check some randomly sampled erroneous reformulations and identify two main reasons why our reformulations do not match the ground truth.

First, some queries are edited to add more information, which is beyond the context of the original query such as “python covert variable to integer” to “python convert hexadecimal to integer” and “git commit -am” to “git commit -am vs git add”. Although SEQUER can successfully revise the misspelling “convert” to “convert”, it cannot guess the replacement of “variable” with “hexadecimal” or adding “vs git add” by considering only the local context of the query. To support such complicated reformulation, we need to consider the broader context of the search (e.g., previous queries) in the future.

Second, the same meaning may be expressed in different ways. For example, given the original query “remove , from input”, SEQUER recommends the reformulated query as “remove comma from input”, however, the ground truth is “remove , from input”. Similarly, given the original query “read mouse cursor”, SEQUER recommends the reformulated query as “@# read mouse cursor”, however, the ground truth is “read mouse cursor in C#”. Although the users’ reformulation results and our recommendations are not exactly matched, they convey the same meaning. Some of our recommendations are of higher quality than the users’ reformulation and may lead to better search results. That is also why the performance of SEQUER is highly underestimated.

E. Discussion

In Section V-D we evaluate the quality of reformulations from different approaches by comparing them with users’ manual ones with the metrics used in the GEC task. However, none of these metrics consider the model effectiveness (i.e., the ability of the reformulated query to retrieve the desired post). Therefore, we further evaluate the retrieval effectiveness of SEQUER in this section.

For each query reformulation thread (mentioned in Section [1]), we collect pairs of users’ original query and finally-visited post, which is assumed to be the target post. These query-post pairs can be used to demonstrate the retrieval effectiveness of the model. For each pair, we adopt both our approach and baselines in Section V-B to reformulate the query and check the ranking of the target post in all candidate posts on Stack Overflow. All the 65,103 queries in the testing set are paired with their corresponding finally-visited post as the evaluation data.

We use MRR (Mean Reciprocal Rank) as the metric for retrieval effectiveness evaluation. MRR is the average of the reciprocal ranks (i.e., the multiplicative inverse of the target post’s rank in the search result) of the search results for all the queries. For example, given a query, if the target post ranks fifth in the search result, the reciprocal rank is 1/5=0.2. A higher MRR value indicates a better search result.

The comparison results between SEQUER and baselines in terms of MRR can be found in Table VI. SEQUER achieves the best performance (i.e., 129.33% boost in terms of MRR to the original query). Even compared with the best baseline seq2seq+Attn, SEQUER still achieves a 23.7% boost. This result shows that the reformulated query given by SEQUER is not only the closest to manual reformulation, but also has the best retrieval effectiveness among all the baselines. Therefore, SEQUER can effectively help users obtain better search results via high-quality query reformulation.

VI. RELATED WORK

Information Retrieval (IR) has been widely used in software engineering (SE) tasks, such as traceability recovery [56], [57].
feature location [58], library migration [2], API search [62, 63] and GUI design seeking [64–66]. In this section, we summarize the related works about query reformulation in general IR and its application in SE domain.

A. Query reformulation in general information retrieval

To help users better refine their queries, there are many studies on query expansion [67, 68], query reformulation [69, 70] and query suggestion [47–50] in information retrieval. In detail, Jiang et al. [48] tried to provide query suggestions based on previous queries in the session by introducing a binary classifier and an RNN-based decoder as the query discriminator and the query generator. Chen et al. [49] proposed an attention-based hierarchical neural query suggestion model that combines a session-level neural network and a user-level neural network to model the users’ short and long term search history.

However, most of these studies are session-based or user profile-based and apply to general text search. Different from general text search, search in the software engineering domain is very specific, with domain-specific terms and code snippets, which make general approaches not applicable in this scenario. Therefore, we carry out this study based on domain-specific dataset for providing a software-specific automated query reformulator.

B. Query reformulation for document search in SE

The performance of document search in software engineering relies on the domain-specific query reformulation. Haiduc et al. proposed several metrics to measure query difficulty [71], query specificity [72], and query quality [73] for concept location. Based on these metrics, they [74, 75] further developed a machine learning model to adopt one of four strategies to recommend revised queries. Rahman et al. [76] proposed a word-embedding based method to extract semantically similar terms from questions on Stack Overflow, hence suggesting semantically relevant queries. Li et al. [77] shared a similar idea by building a software-specific domain lexical database based on tags on Stack Overflow and optimized the input queries to help search software-related documents. Chen et al. [78] reformulated the Chinese queries to English ones for searching related posts on Stack Overflow.

Most of these previous studies generate query reformulation based on heuristic rules or lexical databases, which depends greatly on the quality and size of the rules or the database. In contrast, SEQUER is fully data-driven, which is based on large-scale real-world developers’ queries on Stack Overflow. We believe that SEQUER can automatically learn reformulation patterns, and generate better reformulation result.

C. Query reformulation for code search in SE

Code search plays an important role in software engineering, many previous studies focused on query reformulation for code search. Sisman et al. [30] proposed a query reformulation framework by enriching the users’ queries with certain specific terms drawn from the highest-ranked retrieved artifacts. Howard et al. [79] leveraged similar word pairs in comments and method signatures, which are semantically similar in software engineering to reformulate users’ queries. Lu et al. [68] took a similar method, i.e., identifying each term in the original query and extends with synonyms generated from WordNet. Nie et al. [67] identified software-specific expansion words from high-quality pseudo relevance feedback question and answer pairs on Stack Overflow. Rahman et al. [80] identified terms from the source code using a novel term weight-CodeRank, and then suggested effective reformulation of the original query by exploiting the source document structures, query quality analysis, and machine learning. They further proposed a query reformulation technique that suggests a list of relevant API classes for a natural language query by exploiting keyword-API associations from the questions and answers on Stack Overflow [70]. Similarly, Sirres et al. [81] augmented the original query with structural code entities by mining questions and answers from Stack Overflow.

Different from code search, our study mainly focuses on software-specific document search. Moreover, SEQUER can complement these code search approaches to reformulate the users’ queries better.

VII. Conclusion

Constructing an efficient query to search through a large amount of programming knowledge is a challenging task for developers, especially for novices. Our empirical study on a large scale real-world query records on Stack Overflow indicates that developers always reformulate their queries to obtain the desired results. To assist with developers’ efficient search, we propose a deep learning-based approach SEQUER to learn query reformulation patterns from query logs provided by Stack Overflow. Given the original query, it can automatically recommend a list of reformulation candidates for selection. Evaluation on large-scale archival query reformulations verifies the superiority of SEQUER compared with five state-of-the-art baselines.

In the future, we will further improve the performance of SEQUER by incorporating more contextual information such as the users’ profile, query history, and their post visiting history. Moreover, we plan to take our approach one step further by directly recommending posts for the query. Based on users’ queries and corresponding clicked posts provided by Stack Overflow, we could develop a domain-specific model to learn the relationships between them.

ACKNOWLEDGEMENT

The authors would like to thank Stack Exchange Inc. for sharing the dataset, and the anonymous reviewers for their insightful comments and suggestions. This work is supported in part by the National Natural Science Foundation of China (Grant Nos. 61872263, 61702041 and 61202006), the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University (Grant No. KFKT2019B14), and the Australian Research Council (DE180100153).