Self-Supervised Query Reformulation for Code Search

Yuetian Mao*
Shanghai Jiao Tong University
Shanghai, China
mytkeroro@sjtu.edu.cn

Chengcheng Wan*
East China Normal University
Shanghai, China
wancc1995@gmail.com

Yuze Jiang
Shanghai Jiao Tong University
Shanghai, China
jyz-1201@sjtu.edu.cn

Xiaodong Gu†
Shanghai Jiao Tong University
Shanghai, China
xiaodong.gu@sjtu.edu.cn

ABSTRACT

Automatic query reformulation is a widely utilized technology for enriching user requirements and enhancing the outcomes of code search. It can be conceptualized as a machine translation task, wherein the objective is to rephrase a given query into a more comprehensive alternative. While showing promising results, training such a model typically requires a large parallel corpus of query pairs (i.e., the original query and a reformulated query) that are confidential and unpublished by online code search engines. This restricts its practicality in software development processes. In this paper, we propose *SSQR*, a self-supervised query reformulation method that does not rely on any parallel query corpus. Inspired by pre-trained models, *SSQR* treats query reformulation as a masked language modeling task conducted on an extensive unannotated corpus of queries. *SSQR* extends T5 (a sequence-to-sequence model based on Transformer) with a new pre-training objective named *corrupted query completion* (CQC), which randomly masks words within a complete query and trains T5 to predict the masked content. Subsequently, for a given query to be reformulated, *SSQR* identifies potential locations for expansion and leverages the pre-trained T5 model to generate appropriate content to fill these gaps. The selection of expansions is then based on the information gain associated with each candidate. Evaluation results demonstrate that *SSQR* outperforms unsupervised baselines significantly and achieves competitive performance compared to supervised methods.

CCS CONCEPTS

• **Information systems → Query reformulation**.

KEYWORDS

Query Reformulation, Code Search, Self-supervised Learning

ACM Reference Format:
Yuetian Mao, Chengcheng Wan, Yuze Jiang, and Xiaodong Gu. 2023. Self-Supervised Query Reformulation for Code Search. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

*Both authors contributed equally to this research.
†Xiaodong Gu is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Searching through a vast repository of source code has been an indispensable activity for developers throughout the software development process [49]. The objective of code search is to retrieve and reuse code snippets from existing projects that align with a developer's intent expressed as a natural language query [50]. However, it has been observed that developers often struggle to articulate their information needs optimally when submitting queries [13, 28]. This difficulty may arise from factors such as inconsistent terminology used in the query or a limited understanding of the specific domain in which information is sought. Developers may constantly reformulate their queries until the queries reflect their real query intention and retrieve the most relevant code snippets. Studies [8] have shown that in Stack Overflow, approximately 24.62% of queries on Stack Overflow have undergone reformulation. Moreover, developers, on average, reformulate their queries 1.46 times before selecting a particular result to view.

One common solution to this problem is automatic query reformulation, namely, rephrasing a given query into a more comprehensive alternative [18, 32]. A natural first way to accomplish this objective is to replace words in a query with synonyms based on external knowledge such as WordNet and thesauri [20, 31, 34, 51]. However, this methodology restricts the expansion to the word level. Besides, gathering and maintaining domain knowledge is usually costly. The knowledge base might always lag behind the fast-growing code corpora. There have been other attempts that consider pseudo-relevance feedback, i.e., emerging keywords in the initial search results [19, 22, 43, 53]. They search for an initial set of results using the original query, select new keywords from the top k results using TF-IDF weighting, and finally expand the original query with the emerging keywords. Nevertheless, despite expanding queries at a word level, this approach also has a risk of expanding queries with noisy words. Hence, the expanded query can be semantically irrelevant to the original one.

In recent years, driven by the prevalence of deep learning, researchers seek the idea of casting query reformulation as a machine translation task: the original query is taken as input to a neural sequence-to-sequence model and is translated into a more comprehensive alternative [8]. Despite showing substantial gains, such models require to be trained on a large-scale parallel corpus of query pairs (i.e., the original query and a reformulated query). Unfortunately, acquiring large query pairs is infeasible given that real-world search engines (e.g., Google and Stack Overflow) do not publicly release the evolution of queries. For example, the state-of-the-art method SEQUER [8] relies on a confidential parallel

dataset that cannot (likely to be impossible) be accessed by external researchers. Replicating the performance of SEQUER becomes challenging or even impossible for those who lack access to such privileged datasets. This lack of replicability hampers the wider adoption and evaluation of the method by the research community.

In this paper, we present *SSQR*, a self-supervised query reformulation method that achieves competitive performance to the state-of-the-art supervised approaches, while not relying on the availability of parallel query data for supervision. Inspired by the pre-trained models, *SSQR* automatically acquires the supervision of query expansion through self-supervised training on a large-scale corpus of code comments. Specifically, we design a new pre-training objective called *corrupted query completion* (CQC) to simulate the query expansion process. CQC masks keywords in long, comprehensive queries and asks the model to predict the missing contents. In such a way, the trained model is encouraged to expand incomplete queries with keywords. *SSQR* leverages T5 [42], the state-of-the-art language model for code. The methodology of *SSQR* involves a two-step process. Firstly, T5 is pre-trained using the CQC objective on a vast unannotated corpus of queries. This pre-training phase aims to equip T5 with the ability to predict masked content within queries. When presented with a query to be reformulated, *SSQR* enumerates potential positions within the query that can be expanded. It then utilizes the pre-trained T5 model to generate appropriate content to fill these identified positions. Subsequently, *SSQR* employs an information gain criterion to select the expansion positions that contribute the most valuable information to the original query, resulting in the reformulated query.

We evaluate *SSQR* on two search engines through both automatic and human evaluations, and compare with state-of-the-art approaches, including SEQUER [8], NLP2API [43], LuSearch [34], and GooglePS [11]. Experimental results show that *SSQR* improves the MRR score by over 50% compared with the unsupervised baselines and gains competitive performance over the fully-supervised approach. Human evaluation reveals that our approach can generate more natural and informative queries, with improvements of 19.31% and 26.35% to the original queries, respectively.

Our contributions are summarized as follows:

- To the best of our knowledge, *SSQR* is the first self-supervised query reformulation approach, which does not rely on a parallel corpus of reformulations.
- We propose a novel information gain criterion to select the pertinent expansion positions that contribute the most valuable information to the original query.
- We perform automatic and human evaluations on the proposed method. Quantitative and qualitative results show significant improvements over the state-of-the-art approaches.

2 BACKGROUND

2.1 Code Search

Code search is a technology to retrieve and reuse code from pre-existing projects [9, 16, 50]. Similar to general-purpose search engines, developers often encounter challenges when attempting to implement specific tasks. In such scenarios, they can leverage a code search engine by submitting a natural language query. The search engine then traverses an extensive repository of code snippets collected from various projects, identifying code that is semantically relevant to the given query. Code search can be broadly classified into two categories: search within the context of a specific project or as an open search across multiple projects. The search results may include individual code snippets, functions, or entire projects.

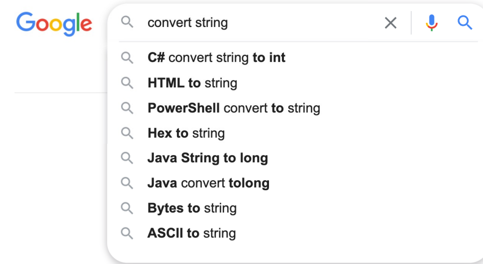

Figure 1: An example of Google query reformulation.

2.2 Query Reformulation

Query reformulation provides an effective way to enhance the performance of search engines [10, 31, 53]. The quality of queries is often a bottleneck of search experience in web search [10]. This is because the initial query entered by the user is often short, generic, and ambiguous. Therefore, the search results could hardly meet the specific intents of the user. This requires the user to revise his query through multiple rounds. Query reformulation is a technology that reformulates user's queries into more concrete and comprehensive alternatives [21]. Figure 1 shows an example of query reformulation in Google search engine. When a user enters the query "convert string" in the search box, there may exist multiple possible intents, such as "convert something to a string" or "convert a string to something". Additionally, the specific programming language for implementing the conversion function is not specified. In such cases, conventional search engines like Google face challenges in accurately determining the user's true intent.

To address this issue, search engines often employ tools like the Google Prediction Service (GooglePS). GooglePS automatically suggests multiple reformulations of the original query. These reformulations provide alternative options that the user can consider to refine their search. By presenting a range of reformulations, users can narrow down their search target by selecting the most relevant reformulation that aligns with their intended query. This process helps users in finding more precise and tailored search results.

Query reformulation broadly encompasses various techniques, including query expansion, reduction, and replacement [25]. While query expansion involves augmenting the original query with additional information, such as synonyms and related entities, to enhance its content, query reduction focuses on eliminating ambiguous or inaccurate expressions. Query replacement, on the other hand, involves substituting incorrect or uncommon keywords in the original query with more commonly used and precise terms. Among these types, query expansion constitutes the predominant approach, accounting for approximately 80% of real-world search scenarios [45].

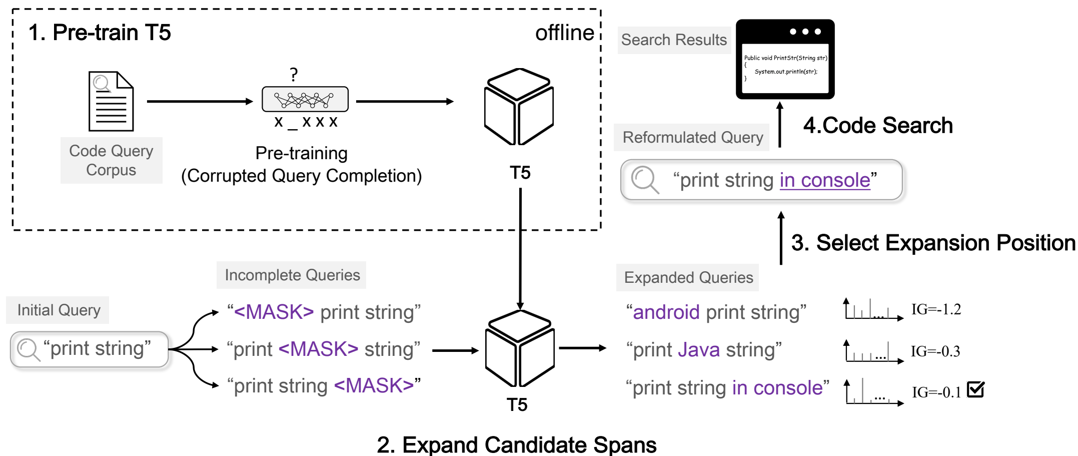

Figure 2: An illustration of the main pipeline

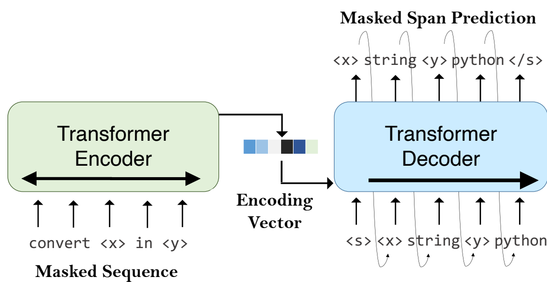

Figure 3: Illustration of T5

2.3 Self-Supervised Learning and Pre-trained Models

Supervised learning is a class of machine learning methods that train algorithms to classify data or predict outcomes by leveraging labeled datasets. It is known to be expensive in manual labeling, and the bottleneck of data annotation further causes generalization errors [24], spurious correlations [29], and adversarial attacks [37]. Self-supervised learning alleviates these limitations by automatically mining supervision signals from large-scale unsupervised data using auxiliary tasks [33]. This enables a neural network model to learn rich representations without the need for manual labeling [52]. For example, the cloze test masks words in an input sentence and asks the model to predict the original words. In this way, the model can learn the semantic representations of sentences from large unlabeled text corpora.

Pre-trained language models (PLMs) such as BERT [12], GPT [7], and T5 [42] are the most typical self-supervised learning technology. A PLM aims to learn language's generic representations on a large unlabeled corpus and then transfer them to specific tasks through fine-tuning on labeled task-specific datasets. This requires the model to create self-supervised learning objectives from the unlabeled corpora. Take the Text-to-Text Transfer Transformer (T5) [42] in Figure 3 as an example. T5 employs the Transformer [47] architecture where an encoder accepts a text as input and outputs the encoded vector. A decoder generates the target sequence based on the encodings. To efficiently learn the text representations, T5 designs three self-supervised pre-training tasks, namely, masked span prediction, masked language modeling, and corrupted sentence reconstruction. By pre-training on large-scale text corpora, T5 achieves state-of-the-art performance in a variety of NLP tasks, such as sentence acceptability judgment [54], sentiment analysis [41], paraphrasing similarity calculation [39], and question answering [27].

3 METHOD

The primary focus of this paper is on query expansion, the most typical (accounting for 80%) technique for query reformulation. Query expansion aims to insert key phrases into a query thereby making it more specific and comprehensive. Essentially, query expansion addresses a *pinpoint-then-expand* problem, wherein the goal is to identify potential information gaps within a given query and generate a set of keywords to fill those gaps.

Inspired by the masked language modeling (MLM) task introduced by pre-trained models like BERT [12], our proposed method adopts a self-supervised idea. Specifically, we mask keywords within complete code search queries and train a model to accurately predict and recover the masked information. This allows the model to learn the underlying patterns and relationships within the queries, enabling it to generate meaningful expansions for query reformulation.

3.1 Overview

Figure 2 shows the main framework as well as the usage scenario of our method. The pipeline involves two main phases: an offline pre-training phase and an online expansion phase. During the pre-training phase, *SSQR* continually pre-trains a PLM named T5 [42] with a newly designed *corrupted query completion* task on an unlabelled corpus of long queries (§3.2). This enables the T5 to learn how to expand incomplete queries into longer ones. During the runtime of *SSQR*, when a user presents a query for code search,

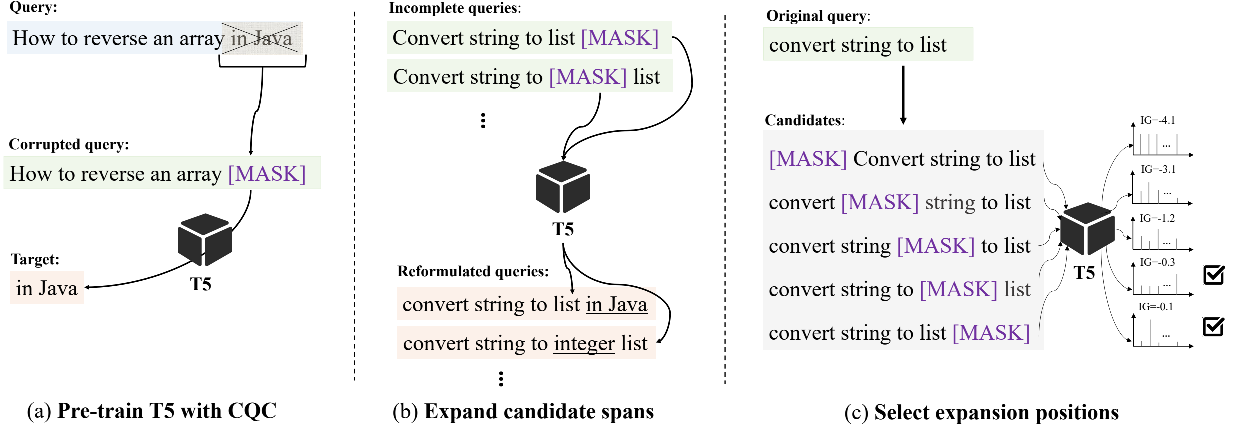

Figure 4: A working example of each expansion step

SSQR employs a two-step process for query expansion. Firstly, it enumerates candidate positions within the query that can be expanded and utilizes the pre-trained T5 model to generate content that fills these positions (as discussed in §3.3). Following the expansion step, *SSQR* proceeds to select the position that offer the highest information gain after the expansion (introduced in §3.4). This selection process ensures that the most valuable and informative expansions are chosen, thereby enhancing the reformulated query in terms of its relevance and comprehensiveness.

Finally, once the query has been expanded, users conduct code search by selecting the most relevant reformulation that aligns with their intended query. Our approach specifically focuses on the function-level code search scenario, which involves the retrieval of relevant functions from a vast collection of code snippets spanning multiple projects.

The following sections elaborate on each step of our approach respectively.

3.2 Pre-training T5 with Corrupted Query Completion

We start by pre-training a PLM which can predict the missing span in a query. We take the state-of-the-art T5 [42] as the backbone model since it has a sequence-to-sequence architecture and is more compatible with generative tasks. Besides, T5 is specialized in predicting masked spans (i.e., a number of words).

To enable T5 to learn how to express a query more comprehensively, we design a new pre-training objective called *corrupted query completion* (CQC) using a large-scale corpus of unlabelled queries. Similar to the MLM objective, CQC randomly masks a span of words in the query and asks the model to predict the masked span. More specifically, given an original query $q = (w_1, \ldots, w_n)$ that consists of a sequence of n words, *SSQR* masks out a span of 15%×n consecutive words from a randomly selected position i, namely, $s_{i:j}=(w_i, \ldots, w_j)$, and replaces it with a [MASK] token. Then, the corrupted query is taken as input to T5 which predicts the words in the masked span. We use the teacher-forcing strategy for pre-training. When predicting a word in the corrupted span, the context visible to the model consists of two parts: 1) the uncorrupted words in the original query, denoted as $q_{\backslash s_{i:j}} = (w_1, \ldots, w_{i-1}, w_{j+1}, \cdots, w_n)$; and 2) the ground truth words appeared before the current predicting position w_t, denoted as $w_{i:t-1}$. We pre-train the model using the cross-entropy loss, namely, minimizing

$$\mathcal{L}_{\text{cqc}} = -\sum_{t=i}^{j} \log p\left(w_t \mid q_{\backslash s_{i:j}}, w_{i:t-1}\right). \tag{1}$$

Figure 4(a) shows an example of the CQC task. For a query “how to reverse an array in Java” taken from the training corpus, the algorithm corrupts the query by replacing the modifier “in Java” with [MASK]. The corrupted query is taken as input to T5 which predicts the original masked tokens “in Java”.

3.3 Expanding Candidate Spans

The pre-trained T5 model is then leveraged to expand queries. We consider a query that needs to be expanded as an incomplete query where a span of words is missing at a position (denoted as a masked token), we want the model to generate a sequence of words to fill in the span. This is exactly the problem of *masked span prediction* as T5 aims to solve. Therefore, we leverage the pre-trained T5 to expand the incomplete queries.

However, a query with n words have n+1 positions for expansion. Therefore, we design a *best-first* strategy: we enumerate all the n+1 positions as the masked spans, perform the CQC task, and select the top-k positions that have the most information gain of predictions. Specially, given a original query $q=\{w_1, w_2, ..., w_n\}$, *SSQR* enumerates the n+1 positions between words. For each position, it inserts a [MASK] token. This results in n+1 candidate masked queries. Each incomplete query $\tilde{q}=[w_1, \ldots, \text{[MASK]}, \ldots, w_n]$ is taken as input to the pre-trained T5. The decoder of T5 generates a span of words $s=[v_1, \ldots, v_m]$ for the [MASK] token by sampling words according to the predicted probabilities. Finally, *SSQR* replaces the [MASK] token with the generated span s, yielding the reformulated query.

Algorithm 1: Span Expansion and Selection

```
Input: q: the input query written in natural language;
       T5: the pre-trained T5 model;
       k: the number of candidate positions;
       m: the maximum target length.
Output: Q: a set of candidate masked queries.
1  w_1, ..., w_n = tokenize(q);
2  Q = {};                                  ▷ initialize the query set.
3  for i = 1 to n do
4      q̃ = w_1, ..., w_i, [MASK], w_{i+1}, ..., w_n;      ▷ insert a
5                          [MASK] token to the i-th position in q.
6      v_1, ..., v_m = T5(q̃);              ▷ predict the content for the
7                                                      [MASK] token.
8      IG = (1/m) Σ_{j=1}^{m} p(v_j)log p(v_j);            ▷ calculate the
9                          information gain for the expansion.
10     Q = Q ∪ ⟨q̃, IG⟩;
11 end
12 Sort Q based on entropy in an descending order;
13 Q= top(Q, k);                            ▷ select the top-k queries.
14 return Q
```

Figure 4(b) shows an example. Given the first two masked queries, the T5 model generates "in Java" and "integer" for the masked tokens, respectively. The former refers to the language used to implement the function, while the latter refers to the data type of the target data structure. The reformulated queries supplement the original queries with additional information, revealing user potential intents from different aspects.

3.4 Selecting Expansion Positions

A query with n words has n+1 candidate positions for expansion, but not all of them are necessary for expansion. Hence, we must determine which positions are the most proper to be expanded. *SSQR* selects the top-k candidate queries that have the most missing information in the masked span. The resulting expanded queries are more likely to gain information after span filling.

The key issue here is how to measure the information gain after filling each span. In our approach, we define *information gain* for a span expansion as the negative entropy over the predicted probability distribution of the generated words [26, 36]. In information theory, entropy [44] characterizes the uncertainty of an event in a system. Suppose the probability that an event will happen follows a distribution of $(p_1, \dots, p_n)$, the entropy of the event can be computed as $-\sum_{i=1}^{n} p_i \log p_i$. The lower the entropy, the more certain that the event can happen. That means the event brings more information to humans. This can be analogized to the span prediction problem: when the probability distribution of the generated words over the vocabulary is uniform, the entropy (uncertainty) becomes high because every word is likely to be generated. By contrast, smaller entropy means that there is a greatly different likelihood of generating each word and thus the certainty of the generation is high. The lower the entropy, the higher the certainty that this span contains the word, and the more information that the expansion brings to the query. If the span contains multiple words, we can measure the information gain of the span prediction using their average negative entropy.

For each candidate query $\tilde{q}$, we predict a span s=$[v_1, ..., v_m]$ using the pretrained T5 model:

$$p(v_i) = T5(\tilde{q}, v_{<i}), i = 1, \dots, L \tag{2}$$

where each v_i denotes a sub-token in the predicted span. Each prediction $p(v_i)$ follows a probability distribution of $p_1, \dots, p_{|V|}$ over the vocabulary of the entire set of queries in the training corpus, indicating the likelihood of each token in the vocabulary appearing in the span.

Our next step is to compute the information gain of each expansion using negative entropy: For each sub-token v_i, the information gain can be calculated by

$$IG(v_i) = -H(v_i) = \sum_{v=1}^{|V|} p(v_i = v)\log p(v_i = v). \tag{3}$$

The higher the IG, the more certainty that the prediction is. For a predicted span $s = [v_1, \dots, v_m]$ with m sub-tokens, we compute the average IG of all its tokens, namely,

$$IG(s) = \frac{1}{m}\sum_{i=1}^{L} IG(v_i). \tag{4}$$

Finally, we select the top K expansions with the highest information gain and then replace the [MASK] token with the predicted span. The top-k expansions are provided to users for choosing the most relevant one that aligns with their intention. The specific details of the method are summarized in Algorithm 1.

Figure 4(c) shows an example query expansion. For a given query "convert string to list" to be expanded, *SSQR* firstly enumerates five expansion positions of the original query, inserting a [MASK] token into each one. Next, the pre-trained T5 model takes these candidate queries as input and calculates the information gain from the prediction for these candidate queries. Finally, *SSQR* recommends the top-2 masked queries (here k=2) with the highest information gain (i.e., minimum entropy values) for users to choose.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We evaluate the performance of *SSQR* in query reformulation through both automatic and human studies. We further explore the impact of different configurations on performance. In specific, we address the following research questions:

- **RQ1: How effective is *SSQR* in query reformulation for code search?**
  We apply query reformulation to Lucene and CodeBERT based code search engines and compare the search accuracy before and after query reformulation by various approaches.
- **RQ2: Whether the queries reformulated by *SSQR* are more contentful and easy to understand?**
  In addition to the automatic evaluation of code search performance, we also want to assess the intrinsic quality of the reformulated queries. To this end, we perform a human study

Table 1: Statistics of Datasets

Stage	Dataset	# of Samples
Pre-training	CODEnn [16]	1,000,000
Fine-tuning	CodeXGLUE [35]	251,820
Search	CodeXGLUE [35]	19,210

to assess whether the reformulated queries contain more information than the original ones and meanwhile conform to human reading habits.

- **RQ3: How do different configurations impact the performance of *SSQR*?**
  To obtain a better insight into *SSQR*, we investigate the performance of *SSQR* under different configurations. We firstly investigate the effect of different positioning strategies, i.e., what is the best criteria to select the expansion position in the original query. We are also interested in the number of expansions for each query.

4.2 Datasets

We pre-train, fine-tune, and test all models using two large code search corpora: CODEnn [16] and CodeXGLUE [35]. They are non-overlapping and thus alleviate the duplicated code issue between pre-training and downstream tasks [6].

Dataset for Pre-training. We pre-train T5 using code comments from the large-scale CODEnn dataset. CODEnn has been specifically processed for code search. Compared to CodeSearchNet [4], this dataset has a much larger volume (i.e., more than 10 million queries). We take the first 1 million for pre-training.

Dataset for Code Search. We use the code search dataset of CodeXGLUE which provides queries and the corresponding code segments from multiple projects. In this dataset, each record has five attributes, including the code segment (in Python), repository URL, code tokens, doc string (i.e., NL description of the function), and the index of the code segment. We split the original dataset into training and test sets, with 251,820 and 19,210 samples respectively. The training set is used to fine-tune the CodeBERT search engine, while the test set is used as the search pool from which a search engine retrieves code. All queries in the test phase are tests on the same pool. The statistics of our datasets are summarized in Table 1.

4.3 Implementation Details

Implementation of the pre-trained model. Our model is implemented based on `T5-base` from the open-source collection of HuggingFace [23]. We use the default tokenizer and input-output lengths. Since HuggingFace does not provide an official PyTorch script for T5 pre-training, we implement the pre-training script based on the PyTorch Lightening framework [5]. We initialize T5 with the default checkpoint provided by Huggingface and continually pre-train it with our proposed CQC task. The pre-training takes 3 epochs with a learning rate of 1e-3. The batch size is set to 32 in all experiments. We set m and k in Algorithm 1 to 10 and 3 respectively. Since T5 is non-deterministic, *SSQR* can generate different queries for an input query at each run. To guarantee the same output at each run, we fix the random seeds to 101 and reload the same *state-dict* of T5.

Implementation of the search engines. We experiment under two search engines based on CodeBERT [14] and Lucene [1].

1) *A CodeBERT-based search engine*: As our approach is built on pre-trained models, we first verify the effectiveness on a pre-training based search engine. Specifically, we test our approach on the default search engine by CodeBERT. We reuse the implementation of the code search (i.e., Text-Code) task in CodeXGLUE [35]. Then, we fine-tune the CodeBERT-base checkpoint on a training set from CodeXGLUE for 2 epochs with a constant learning rate of 5e-5.

2) *The Lucene search engine*: Besides the pre-training based search engine, we also test our approach on a classic search engine named Lucene [1]. Lucene is a keyword-based search library that is widely adapted to a variety of code search engines and platforms [2, 3, 15]. We implement the Lucene search engine based on the Lucene core in Java. We extract the code segments from the test dataset from CodeXGLUE, parse them by Lucene's StandardAnalyzer, and build their indexes.

We train all models on a Linux server with Ubuntu 18.04.1 and a GPU of Nvidia GeForce RTX 2080 Ti.

4.4 Baselines

We compare our method with the state-of-the-art query reformulation approaches, including a supervised method called SEQUER, and unsupervised methods such as NLP2API, LuSearch, and GooglePS.

1) **Supervised** [8]: a supervised learning approach for query reformulation named SEQUER. SEQUER leverages Transformer [47] to learn the sequence-to-sequence mapping between the original and the reformulated queries. The method relies on a confidential parallel dataset of query evolution logs provided by Stack Overflow. The dataset contains internal HTTP requests processed by Stack Overflow's web servers within one year.
2) **NLP2API** [43]: a feedback based approach that expands query with recommended APIs. NLP2API automatically identifies relevant API classes collected from Stack Overflow using keywords in the initial search results and then expands the query with these API classes.
3) **LuSearch** [34]: a knowledge-based approach that expands a query with synonyms in WordNet [38]. The reformulated queries by LuSearch are based on Lucene's structural syntax, which are too long and contain too many Lucene-specific keywords. Due to the constraint on the input length of T5 model (i.e., 512 tokens), we keep the synonyms and remove keywords about attribute names in Lucene such as "methbody" and "methname".
4) **GooglePS** [11]: the Google query prediction service that gives real-time suggestions on query reformulation. We directly enter test queries into the Google search box and manually collect the reformulated queries in RQ2 and the case study. We do not compare our method with GooglePS in RQ1 because its search API is unavailable to us for processing a large number of queries. Besides, our baseline model

has demonstrated a great improvement over it in terms of MRR [8].

4.5 Evaluation Metrics

The ultimate goal of query reformulation is to enhance search accuracy by using the reformulated queries. In our experiments, we first evaluate the search accuracy measured by the widely used mean reciprocal rank (MRR). MRR is defined as the average of the reciprocal ranks (i.e., the multiplicative inverse of the target post's rank) of the search results for all the queries, namely,

$$MRR = \frac{1}{Q} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{5}$$

where Q refers to a set of queries and $rank_i$ stands for the position of the first relevant document for the i-th query. A higher MRR indicates better search performance.

Besides the indirect criteria in search performance, query reformulation also aims to help users write more precise and high-quality queries. Therefore, we further define two metrics to measure the intrinsic quality of the reformulated queries:

- *Informativeness* measures how much information a query contains that contributes to code search. We use this metric to evaluate how much information gain the reformulation brings to the original query.
- *Naturalness* measures how well a query is grammatically correct and follows human reading habits. By using this metric, we want the reformulation to be semantically coherent with the original query.

Both metrics range from 1 to 5. Higher scores indicate better performance.

5 RESULTS

5.1 RQ1: Performance on Code Search

As the ultimate goal of query reformulation, we first evaluate whether the reformulated queries by *SSQR* lead to better code search performance. We experiment under both search engines and compare the improvement of MRR scores before and after query reformulation by various methods. For each query, we calculate its similarity to code instances in the test set. The top 100 instances with the highest similarity are selected as the search results. Each query has one ground-truth code instance in the test set. We calculate the MRR scores by comparing the results and the ground-truth code. Then, for each method, we select the first three reformulations and report the highest MRR score among them. Since the purpose of query reformulation is to hit the potential search intent of the user. We believe that results with the maximum MRR in the top-k reformulations are the most likely to satisfy this goal and are therefore considered meaningful.

The experimental results are presented in Table 2. *SSQR* enhances the MRR by 9.90% and 12.23% on the two search engines, respectively. Compared to the two unsupervised baselines, LuSearch and NLP2API, it brings a giant leap of over 50% in search accuracy. More surprisingly, *SSQR* achieves competitive results to the supervised counterpart though it is not given with any annotations. This indicates that our self-supervised approach can assist developers to

Table 2: Performance of Various Approaches in Code Search

Search Engine	Approach	MRR
CodeBERT	No Reformulation	0.202
	Supervised[1] [8]	0.222 (+9.90%)
	LuSearch [34]	0.144 (-28.70%)
	NLP2API [43]	0.148 (-26.87%)
	SSQR (ours)	**0.222** (+**9.90%**)
Lucene	No Reformulation	0.133
	Supervised [8]	**0.155** (+**16.91%**)
	LuSearch [34]	0.129 (-2.64%)
	NLP2API [43]	0.136 (+2.87%)
	SSQR (ours)	0.149 (+12.23%)

[1] The result is not reproducible due to the unavailability of the confidential parallel data they use.

write high-quality queries, which ultimately leads to better code search results.

We notice that the performance of *SSQR* is slightly worse than the supervised counterpart with the Lucene search engine. This is probably because the supervised approach applies fixed expansion patterns to queries and therefore tends to expand queries with common, fixed keywords. These keywords can be easily hit by search engines based on keyword matching (e.g., Lucene). On the contrary, *SSQR* does not use fixed expansion patterns and thus has more various keywords. Lucene is not able to perform keyword matching on them. Instead, CodeBERT, which models the semantic relationships between keywords, can understand queries expanded by *SSQR*.

Another interesting point is that LuSearch and NLP2API do not contribute to the CodeBERT-based search engine. This is probably because both approaches append words to the tail of the original query, hence perturbing the semantics of the original query when we use deep learning based search engines such as CodeBERT.

5.2 RQ2: Qualitative Evaluation

To evaluate the intrinsic quality of the reformulated queries, we perform a human study with programmers. Four participants from author's institution, but different labs, are recruited through invitations. All participants are postgraduates in the area of software engineering or natural language processing, having over-four-year programming experience. We took the first 100 queries from the test set in RQ1, and reformulated them using various methods, including SEQUER, LuSearch, NLP2API, *SSQR*, and GooglePS. We assigned 100 search tasks to human annotators using these 100 queries and present the reformulated queries by various approaches. The annotators were asked to search code using Google and provide their ratings (on a scale of 1 to 5) towards the reformulation in terms of informativeness and naturalness, without knowing the source of the reformulation tool.

Table 3 summarizes the quality ratings by annotators. Overall, *SSQR* achieves the most improvement in terms of naturalness (19%) and informativeness (26%), showing that it reformulated queries are more human-like. Comparatively, GooglePS and SEQUER have much less improvement. LuSearch and NLP2API even decrease the

Table 3: Human Evaluation Result

Approach	Naturalness	Informativeness
No Reformulation	3.21	3.15
Supervised [8]	3.63 (+13.08%)	3.44 (+9.21%)
LuSearch [34]	2.63 (-18.07%)	3.17 (+0.63%)
NLP2API [43]	2.80 (-12.77%)	3.50 (+11.11%)
GooglePS [11]	3.27 (+1.87%)	3.33 (+5.71%)
SSQR (ours)	**3.83** (+**19.31%**)	**3.98** (+**26.35%**)

naturalness and informativeness, as they directly append relevant APIs or synonyms to the tail of the original query, and thus break the coherence of the query.

In particular, compared with the strong baseline SEQUER, *SSQR* obtains a greater improvement of 17.14% in terms of informativeness, while outperforming slightly in terms of naturalness. The main reason could be that SEQUER applies three reformulation patterns, i.e., deleting unimportant words, rewriting typos, and adding keywords, where only the last pattern increases the informativeness of the query. Besides, SEQUER often adds keywords in a monotonous pattern, such as appending "in Java" at the tail of the queries; meanwhile, our method can generate diverse spans at the proper positions of the original queries. Consequently, the reformulated queries by our approach are more informative.

5.3 RQ3: Performance under Different Configs

In this experiment, we evaluate the performance of *SSQR* in code search under different configurations with the CodeBERT search engine. We vary the positioning strategy and the number of candidate positions in order to search for the optimal configuration.

Positioning Strategies. Selecting expansion positions is critical to the performance. We compare three strategies, including the entropy-based criterion:

- RAND randomly selects k positions in the original query for expansion.
- PROB selects the top-k positions that have the maximum probability while predicting their missing content.
- ENTR selects the top-k positions that have the minimum entropy while predicting their missing content.

The results are shown in Table 4. The PROB and ENTR strategies bring a large improvement (around 10%) to the code search performance. This indicates that both criteria correctly quantify the missing information at various positions. Between these two strategies, ENTR performs slightly better than PROB, probably because ENTR considers the entire distribution of the prediction while PROB just considers the maximum one. As expected, the RAND strategy causes a degradation of 6.68% in code search performance because it selects expansion positions without any guidance, which results in incorrect or redundant expansions.

Number of Candidate Positions. We also investigate how many expansions lead to the best performance. We vary the number of candidate positions from 1 to 3 and verify their effects on performance. Table 5 shows the results. We observe that increasing the number of candidate positions has a positive effect on performance. The best performance is achieved when 3 candidate positions are expanded. Meanwhile, only one candidate position can have a negative effect on query reformulation. The reason can be that our method reformulates the original query with a variety of query intents. A larger number of candidate positions can hit more user intents and hence leads to better search accuracy.

Table 4: Performance of *SSQR* under Different Positioning Strategies

Strategy	MRR	Improvement
RAND	0.1886	-6.68%
PROB	0.2220	+9.85%
ENTR	**0.2221**	+**9.90**%

Table 5: Performance of *SSQR* under Different Candidate Position Numbers

# Positions	MRR	Improvement
1	0.1726	−14.60%
2	0.2074	+ 2.62%
3	**0.2221**	+ **9.90**%

5.4 Qualitative Analysis

To further understand the capability of *SSQR*, we qualitatively examine the reformulation samples by various methods. Four examples are provided in Table 6. Example 1 compares the reformulation for the query "The total CPU load for the Synology DSM" by various methods. The original query aims to find the code that monitors the CPU load of a DSM. The reformulated query by *SSQR* is more precise to the real scenario since CPU load and memory usage are often important indicators that need to be monitored simultaneously. In contrast, SEQUER only removes the "total" at the beginning of the query during reformulation. LuSearch appends the query with the synonyms of the keyword "total" such as "sum" and "aggreg". Meanwhile, NLP2API appends the query with APIs that are relevant to CPU and operating system, which are more useful compared to those of SEQUER and LuSearch. GooglePS appends the word "7" after "DSM" to indicate the version of DSM, which helps to narrow the range of possible solutions.

In Example 2, the original query "Fetch the events" is incomplete and ambiguous because the user does not specify what events to fetch and where to fetch them from. The reformulated query by *SSQR* is more informative than that by SEQUER: *SSQR* specifies the source of events, i.e., from the server, which makes the query more concrete and understandable; meanwhile, SEQUER only restricts the programming language of the target code, without alleviating the ambiguity of the original query. LuSearch expands the synonyms of "Fetch" such as "get" and "convei" to the tail of the query. NLP2API adds APIs relevant to events to the original query. But these synonyms and APIs have limited effect on improving search accuracy. GooglePS specifies the requirement of the query to be a service by adding "Service worker" at the beginning of the original query. But such a specification has a limited effect on narrowing the search space.

Table 6: Examples of Query Reformulation by Various Methods

Original	The total CPU load for the Synology DSM
SEQUER	The CPU load the Synology DSM
LuSearch	The total CPU load for the Synology DSM **sum total aggreg**
NLP2API	The total CPU load for the Synology DSM **OperatingSystemMXBean ProcCpu String**
GooglePS	The total CPU load for the Synology DSM **7**
SSQR	The total **memory usage and** CPU load for the Synology DSM
Original	Fetch the events
SEQUER	Fetch the events **c++**
LuSearch	Fetch the events **bring get convei**
NLP2API	Fetch the events **CalendarEntry ManyToOne OneToMany**
GooglePS	**Service worker** fetch event
SSQR	Fetch the events **from the server**
Original	Get method that raises MissingSetting if the value was unset.
SEQUER	Get method that raises MissingSetting if the value was unset **in c#**
LuSearch	Get method that raises MissingSetting if the value was unset. **beget get engend**
NLP2API	Get method that raises MissingSetting if the value was unset. **Praveen Kumar Date**
GooglePS	**PHP foreach** unset.
SSQR	Get method that raises **an exception with** MissingSetting if the value was unset.
Original	Load values from a dictionary structure. Nesting can be used to represent namespaces.
SEQUER	Load values from a dictionary structure. Nesting can be used to represent namespaces.
LuSearch	Load values from a dictionary structure. Nesting can be used to represent namespaces. **cargo lade freight**
NLP2API	Load values from a dictionary structure. Nesting can be used to represent namespaces. **Amino TKey TValue**
GooglePS	Python namespace class
SSQR	Load **a list of** values from a dictionary structure. Nesting can be used to represent namespaces.

Example 3 shows the results for the query "Get method that raises MissingSetting if the value was unset." The reformulated query by *SSQR* recognizes that MissingSetting is an exception and prepends it with the exception keyword. This facilitates the search engine to find code with similar functionality. In contrast, SEQUER just specifies the programming language of the target code. Compared to *SSQR* and SEQUER, LuSearch and NLP2API only append irrelevant APIs and synonyms to the original query. Hence, the semantics of the query are broken. GooglePS fails to reformulate such a long query. Instead, it returns a search query from other users that contains the keyword "unset". The returned results by GooglePS discard much information from the original query, making deviate from the user intent.

Finally, the last example shows a worse case. Although *SSQR* achieves the new state-of-the-art, it might occasionally produce error reformulations. *SSQR* prepends a modifier "a list of" in front of the word "values", which conflicts with "dictionary" in the given query and thus hampers the code search performance. This is probably because the word "values" occurs frequently in the training corpus and often refers to elements in arrays and lists. Therefore, *SSQR* tends to expand it with modifiers such as "all the" and "a list of". Comparably, SEQUER does nothing to the original query. LuSearch concerns "Load" as the keyword and expands it. NLP2API adds APIs relevant to the key and value of the dictionary data structure, which results in better search performance. GooglePS cannot handle such a long query and just gives an irrelevant reformulation.

These examples demonstrate the superiority of *SSQR* in query reformulation for code search, affirming the strong ability of both position prediction and span generation. In future work, we will conduct empirical research on the error types, and improve our model for the challenging reformulations.

6 DISCUSSION

6.1 Strength of *SSQR* over fully supervised approaches?

One debatable question is what are the benefits of *SSQR* since it does not beat the SOTA fully-supervised approach in terms of the code search metrics.

Fully-supervised methods such as SEQUER achieve the state-of-the-art performance by sequence-to-sequence learning on a parallel query set. However, acquiring such parallel queries is infeasible since the query evolution log by search engines such as Google and Stack Overflow is not publicly available. Besides, the sequence-to-sequence approach tends to learn generic reformulation patterns, e.g., specifying the programming language or deleting a few irrelevant words.

Compared to SEQUER, *SSQR* does not rely on the supervision of parallel queries, instead, it is trained on a nonparallel dataset (queries only) that does not need to collect the ground-truth reformulations. This significantly scales up the size of training data, and therefore allows the model to learn diverse reformulation patterns from a large number of code search queries. *SSQR* provides an alternative feasible and cheap way of achieving the same performance.

6.2 Limitations and Threats

We have identified the following limitations and threats to our method:

Patterns of query reformulation. In this work, we mainly explore query expansion, the most typical class of query reformulation. While query expansion is only designed to supplement queries with more information, redundant or misspelled words in the query can also hamper the code search performance, which cannot be handled by our method. Thus, in future work, we will extend our approach to support more reformulating patterns, including query simplification and modification. For example, in addition to only inserting a `[MASK]` token in the CQC task, we can also replace the original words with a `[MASK]` token or simply delete a token and ask the pre-trained model to predict the deletion position. A classification model can also be employed to decide whether to add, delete or modify keywords in the original query.

Code comments as queries. As obtaining real code queries from search websites is difficult, we use code comments from code search datasets to approximate code queries in building and evaluating our model. Although code comments are widely used for training machine learning models on NL-PL matching [14, 16, 48], they may not represent the performance of queries in real-world code search engines.

7 RELATED WORK

7.1 Query Reformulation for Code Search

Query reformulation for code search has gained much attention in recent years [19, 20, 34, 43, 46, 51]. There are approximately three categories of technologies, namely, knowledge-based, feedback-based, and deep learning based approaches.

The knowledge-based approaches aim to expand or revise the initial query based on external knowledge such as WordNet [38] and thesauri. For example, Howard et al. [20] reformulated queries using semantically similar words mined from method signatures and corresponding comments in the source code. Satter and Sakib [46] proposed to expand queries with co-occurring words in past queries mined from code search logs. Yang and Tan [51] constructed a software-specific thesaurus named SWordNet by mining code- comment mappings. They expanded queries with similar words in the thesaurus. Lu et al. [34] proposed LuSearch which extends queries with synonyms generated from WordNet.

Unlike knowledge-based approaches, feedback-based approaches identify the possible intentions of the user from the initial search results and use them to update the original query. For example, Rahman and Roy [43] proposed to search Stack Overflow posts using pseudo-relevance feedback. Their approach identifies important API classes from code snippets in the posts using TF-IDF, and then uses the top-ranked API classes to expand the original queries. Hill et al. [19] presented a novel approach to extract natural language phrases from source code identifiers and hierarchically classify phrases and search results, which helps developers quickly identify relevant program elements for investigation or identify alternative words for query reformulation.

Recently, deep learning has advanced query reformulation significantly [8, 30]. Researchers regard query reformulation as a machine translation task and employ neural sequence-to-sequence models. For example, Cao et al. [8] trained a sequence-to-sequence model with an attention mechanism on a parallel corpus of original and reformulated queries. The trained model can be used to reformulate a new query from Stack Overflow.

While deep learning based approaches show more promising results than previous approaches, they rely on the availability of large, high-quality query pairs. For example, Cao et al.'s work requires the availability of query pairs within the same session in the search logs of Stack Overflow. But such logs are confidential and unavailable to researchers. This restricts their practicality in real-world code search.

Unlike these works, *SSQR* is a data-driven approach based on self-supervised learning. *SSQR* expands queries by pre-training a Transformer model with corrupt query completion on large unlabeled data. Results demonstrate that *SSQR* achieves competitive results to that of fully-supervised models without requiring data labeling.

7.2 Code Intelligence with Pre-trained Language Models

In recent years, there is an emerging trend in applying pre-trained language models to code intelligence [14, 17, 40, 48]. For example, Feng et al. [14] pre-trained the CodeBERT model based on the Transformer architecture using programming and natural languages. CodeBERT can learn the generic representations of both natural and programming languages that can broadly support NL-PL comprehension tasks (e.g., code defect detection, and natural language code search) and generation tasks (e.g., code comment generation, and code translation). Wang et al. [48] proposed CodeT5, which extends the T5 with an identifier-aware pre-training task. Unlike encoder-only CodeBERT, CodeT5 is built upon a Transformer encoder-decoder model. It achieves state-of-the-art performance on both code comprehension and generation tasks in all directions, including PL-NL, NL-PL, and PL-PL.

To the best of our knowledge, *SSQR* is the first attempt to apply PLM in query reformulation, which aims to leverage the knowledge learned by PLM to expand queries.

8 CONCLUSION

In this paper, we propose *SSQR*, a novel self-supervised approach for query reformulation. *SSQR* formulates query expansion as a masked query completion task and pre-trains T5 to learn general knowledge from large unlabeled query corpora. For a search query, *SSQR* guides T5 through enumerating multiple positions for expansion and selecting positions that have the best information gain for expansion. We perform both automatic and human evaluations to verify the effectiveness of *SSQR*. The results show that *SSQR* generates useful and natural-sounding reformulated queries, outperforming baselines by a remarkable margin. In the future, we will explore other reformulation patterns such as query simplification and modification besides query expansion. We also plan to compare the performance of our approach with large language models such as GPT-4.

DATA AVAILABILITY

Our source code and experimental data are publicly available at https://github.com/RedSmallPanda/SSQR.

ACKNOWLEDGMENTS

This research is supported by National Natural Science Foundation of China (Grant No. 62232003, 62102244, 62032004) and CCF-Tencent Open Research Fund (RAGR20220129).

REFERENCES

[1] 2001. Lucene. https://lucene.apache.org.
[2] 2006. Apache Solr. https://solr.apache.org.
[3] 2012. Index Tank. https://github.com/LinkedInAttic/indextank-engine.
[4] 2019. CodeSearchNet. https://github.com/github/CodeSearchNet.
[5] 2019. PyTorch Lightning. https://www.pytorchlightning.ai.
[6] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!, October 23-24, 2019.* 143–153.
[7] Tom B. Brown, Benjamin Mann, Nick Ryder, and et. al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33, NeurIPS 2020, December 6-12, 2020.*
[8] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated Query Reformulation for Efficient Search based on Query Logs From Stack Overflow. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, 22-30 May 2021.* 1273–1285.
[9] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-Domain Deep Code Search with Meta Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 487–498.
[10] Jia Chen, Jiaxin Mao, Yiqun Liu, Fan Zhang, Min Zhang, and Shaoping Ma. 2021. Towards a better understanding of query reformulation behavior in web search. In *Proceedings of the Web Conference.* 743–755.
[11] R. C. Cornea and N. B. Weininger. 2014. Providing autocomplete suggestions.
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, June 2-7, 2019, Volume 1.* 4171–4186.
[13] Zachary Eberhart and Collin McMillan. 2022. Generating clarifying questions for query refinement in source code search. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* 140–151.
[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[15] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* O'Reilly Media, Inc.
[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *IEEE/ACM 40th International Conference on Software Engineering, ICSE 2018.* 933–944.
[17] Mohammad Abdul Hadi, Imam Nur Bani Yusuf, Ferdian Thung, Kien Gia Luong, Jiang Lingxiao, Fatemeh H Fard, and David Lo. 2022. On the Effectiveness of Pretrained Models for API Learning. *arXiv preprint arXiv:2204.03498* (2022).
[18] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *35th International Conference on Software Engineering, ICSE 2013, May 18-26, 2013.* 842–851.
[19] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering, ICSE.* 232–242.
[20] Matthew J Howard, Samir Gupta, Lori Pollock, and K Vijay-Shanker. 2013. Automatically mining software-based, semantically-similar words from comment-code mappings. In *10th working conference on mining software repositories, MSR 2013.* 377–386.
[21] Jeff Huang and Efthimis N Efthimiadis. 2009. Analyzing and evaluating query reformulation strategies in web search logs. In *Proceedings of the 18th ACM conference on Information and knowledge management.* 77–86.
[22] Qing Huang, Yangrui Yang, Xudong Wang, Hongyan Wan, Rui Wang, and Guoqing Wu. 2017. Query expansion via intent predicting. *International Journal of Software Engineering and Knowledge Engineering* 27, 09n10 (2017), 1591–1601.
[23] HuggingFace. 2022. T5-base model checkpoint. https://huggingface.co/t5-base.
[24] Daniel Jakubovitz, Raja Giryes, and Miguel RD Rodrigues. 2019. Generalization error in deep learning. In *Compressed sensing and its applications.* Springer, 153–193.
[25] Bernard J Jansen, Danielle L Booth, and Amanda Spink. 2009. Patterns of query reformulation during web searching. *Journal of the american society for information science and technology* 60, 7 (2009), 1358–1371.
[26] Feng Jiang, Yuefei Sui, and Lin Zhou. 2015. A relative decision entropy-based feature selection approach. *Pattern Recognition* 48, 7 (2015), 2151–2163.
[27] Zhengbao Jiang, Jun Araki, Haibo Ding, and Graham Neubig. 2021. How can we know when language models know? on the calibration of language models for question answering. *Transactions of the Association for Computational Linguistics* 9 (2021), 962–977.
[28] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Software Eng.* 32, 12 (2006), 971–987.
[29] Richard A Kronmal. 1993. Spurious correlation and the fallacy of the ratio standard revisited. *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 156, 3 (1993), 379–392.
[30] Dawn Lawrie and Dave Binkley. 2018. On the value of bug reports for retrieval-based bug localization. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME.* 524–528.
[31] Xiangsheng Li, Jiaxin Mao, Weizhi Ma, Zhijing Wu, Yiqun Liu, Min Zhang, Shaoping Ma, Zhaowei Wang, and Xiuqiang He. 2022. A Cooperative Neural Information Retrieval Pipeline with Knowledge Enhanced Automatic Query Reformulation. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining.* 553–561.
[32] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Jiazhan Xie, Huanjun Xu, and Yanjun Yang. 2022. How to formulate specific how-to questions in software development?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 306–318.
[33] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Li Mian, Zhaoyu Wang, Jing Zhang, and Jie Tang. 2023. Self-Supervised Learning: Generative or Contrastive. *IEEE Trans. Knowl. Data Eng.* 35, 1 (2023), 857–876.
[34] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, March 2-6, 2015.* 545–549.
[35] Shuai Lu, Daya Guo, Shuo Ren, and et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).
[36] Pasi Luukka. 2011. Feature selection using fuzzy entropy measures with similarity classifier. *Expert Systems with Applications* 38, 4 (2011), 4600–4607.
[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).
[38] G. A. Miller. 1990. Introduction to WordNet : An Online Lexical Database. In *Meeting of the Association for Computational Linguistics.*
[39] Animesh Nighojkar and John Licato. 2021. Improving paraphrase detection with the adversarial paraphrasing task. *arXiv preprint arXiv:2106.07691* (2021).
[40] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022.* 01–13.
[41] Keval Pipalia, Rahul Bhadja, and Madhu Shukla. 2020. Comparative analysis of different transformer based architectures used in sentiment analysis. In *9th International Conference System Modeling and Advancement in Research Trends, SMART.* IEEE, 411–415.
[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.
[43] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Effective Reformulation of Query for Code Search Using Crowdsourced Knowledge and Extra-Large Data Analytics. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, September 23-29, 2018.* 473–484.
[44] Alfréd Rényi et al. 1961. On measures of entropy and information. In *fourth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Berkeley, California, USA.
[45] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *10th joint meeting on foundations of software engineering.* 191–201.
[46] Abdus Satter and Kazi Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. In *19th International Conference on Computer and Information Technology, ICCIT.* 586–591.
[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[48] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

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

[50] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 344–354.

[51] Jinqiu Yang and Lin Tan. 2014. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering* 19, 6 (2014), 1856–1886.

[52] Shouliang Yang, Xiaodong Gu, and Beijun Shen. 2022. Self-supervised learning of smart contract representations. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*. ACM, 82–93.

[53] Yunchang Zhu, Liang Pang, Yanyan Lan, Huawei Shen, and Xueqi Cheng. 2022. LoL: A Comparative Regularization Loss over Query Reformulation Losses for Pseudo-Relevance Feedback. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 825–836.

[54] Gustavo Zomer and Ana Frankenberg-Garcia. 2021. Beyond Grammatical Error Correction: Improving L1-influenced research writing in English using pre-trained encoder-decoder models. In *Findings of the Association for Computational Linguistics (EMNLP)*. 2534–2540.