# Neural Query Expansion for Code Search

Jason Liu
Facebook, Inc.
U.S.A
jasonliu@fb.com

Seohyun Kim
Facebook, Inc.
U.S.A
skim131@fb.com

Vijayaraghavan Murali
Facebook, Inc.
U.S.A
vijaymurali@fb.com

Swarat Chaudhuri
Rice University
U.S.A
swarat@rice.edu

Satish Chandra
Facebook, Inc.
U.S.A
satch@fb.com

## Abstract

Searching repositories of existing source code for code snippets is a key task in software engineering. Over the years, many approaches to this problem have been proposed. One recent tool called *NCS*, takes in a natural language query and outputs relevant code snippets, often being able to correctly answer Stack Overflow questions. But what happens when the developer doesn't provide a query with a clear intent? What if shorter queries are used to demonstrate a more vague intent?

We find that the performance of *NCS* regresses with shorter queries. Furthermore, data from developers' code search history logs shows that shorter queries have a less successful code search session: there are more query reformulations and more time is spent browsing the results. These observations lead us to believe that using *NCS* alone with short queries may not be productive enough.

In this paper, we explore an additional way of using neural networks in code search: the *automatic expansion of queries*. We present *NQE*, a neural model that takes in a set of keywords and predicts a set of keywords to expand the query to *NCS*. *NQE* learns to predict keywords that co-occur with the query keywords in the underlying corpus, which helps expand the query in a productive way. Our results show that with query expansion, *NQE* + *NCS* is able to perform better than using *NCS* alone.

***CCS Concepts*** **• Software and its engineering** → *Software development techniques.*

***Keywords*** code search, word-embedding, deep learning

*MAPL '19, June 22, 2019, Phoenix, AZ, USA*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6719-6/19/06…$15.00
https://doi.org/10.1145/3315508.3329975

**ACM Reference Format:**
Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural Query Expansion for Code Search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3315508.3329975

## 1 Introduction

Searching repositories of existing source code for relevant code snippets is a key task in software engineering. Over the years, many approaches to this problem have been proposed (see Section 2). Recent work in this area has begun to use neural techniques to get powerful code search performance. For example, Facebook has developed an approach to the problem, called *Neural Code Search* (*NCS*) [31]. *NCS* computes a continuous vector embedding of programs at a method-level granularity. Queries (sets of keywords) are mapped to the same vector space, and vector distance is used to simulate relevance of code fragments to a given query. The appeal of such embedding-based techniques is that they automatically learn semantic and structural features of programs and reduce the burden of feature engineering, which was sometimes needed in earlier work.

Sachdev et al. showed that *NCS* is able to correctly answer a subset of Android Java questions obtained from Stack Overflow. In most of the Stack Overflow questions, the intent is quite clear and provides enough representative words for *NCS* to compute a meaningful vector. However, in the context of code search, this may not always be the case. A developer may not know the full method name to search for, or related words to add to the query. In these scenarios, the developer may use a shorter query to express a more vague intent, in hope that code search will be able offer matches with multiple intents. With shorter queries, are developers still able to achieve an equally successful search experience?

To answer this question, we investigated code search behavior for a week at our company. A code search session includes all events that occurred between opening and closing the code search website. An event is defined to be an action taken by the developer that indicates interest in a

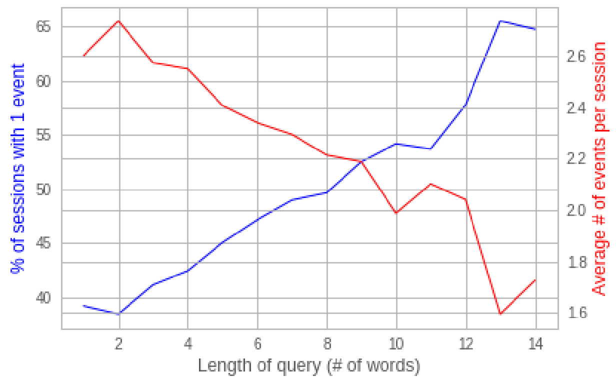

**Figure 1.** Data trends based on code search behavior. Blue line indicates there were more code search sessions with longer queries that only had one event compared to those with shorter queries. Red line indicates code search sessions with shorter queries had more events than those with longer queries.

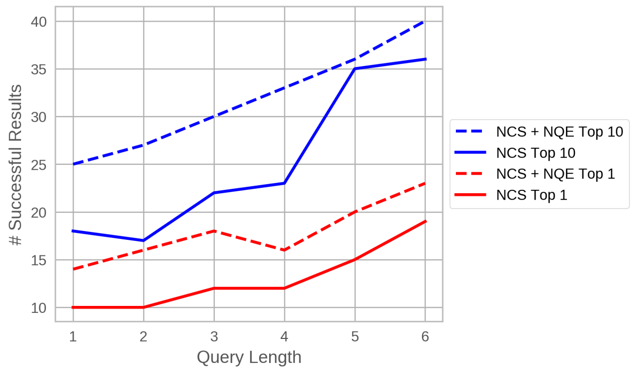

**Figure 2.** Evaluation results on the *Stack Overflow* dataset. Solid lines are results when using *NCS* by itself, and dashed lines indicate using *NCS* and *NQE*.

**Table 1.** Examples of *NQE* improving the *NCS* ranking by expanding the original query with relevant keywords. Queries were manually labeled.

| Original Query | NQE Expansion | NCS Rank | NCS + NQE Rank |
|---|---|---|---|
| get manager package | info | 5 | 1 |
| image get loading | cancel set | 23 | 4 |
| sql | exec on create | 18 | 3 |
| decks | get current | 30 | 2 |
| edit commit | clear | 5 | 1 |

particular search result (e.g. opening the result in a separate browser, copying or selecting text, actively hovering over the result, etc). From the logs, we made two observations.

First, longer queries did not need as much query reformulation as did shorter queries. Query reformulation is defined as the occurrence of a query being altered within the same session, often indicating that a satisfactory result was not found using the initial query. 9.53% of search sessions contained one ore more query reformulations. The average character length of the queries without any change was 23.8, while the average length of the queries with reformulation increased from 20.4 to 23.1. This shows that developers tend to add more characters to the shorter queries to bring them to a length comparable to the length of queries that did not have to be reformulated. Among the queries with reformulation, 33.0% were a subset of the respective final queries, indicating that query *expansion* was quite common.

Second, there was a negative correlation between the length of the query and the number of events for that session. The red line in Figure 1 indicates that for shorter queries, the developer spent more time browsing through the results until ending the search session. On the other hand, the blue line in Figure 1 shows a positive correlation between the length of the query and the percentage of sessions that contained only one event. Sessions with only one event indicate that the developer had a clearer sense of what to look for, rather than spend time exploring the results.

These observations demonstrate a correlation between the lengths of queries and the success of code search. If a tool is able to automatically provide these expanded query words during the search session, a developer could have a more efficient search session, needing less query reformulation. However, these observations were made with a tool that offers a grep-like search, instead of a tool that accepts more free-form search, such as *NCS*. With the availability of such a tool, does having shorter queries regress the performance?

To answer this question, we evaluated *NCS* on the evaluation dataset used in Sachdev et al.[31], varying the length of the queries for this experiment. The evaluation dataset consists of 518 Android-specific queries obtained from Stack Overflow. These questions were chosen using a script that passed several criteria, including having "Android" and "Java" tags, and the accepted code snippet answer having at least one match in a corpus of GitHub Android repositories. For this experiment, we constructed six variations of each query in the following way: after tokenizing the query (splitting words by snake and camel case, removing stop words, removing non-alphanumerical characters), we chose top $n$ TF-IDF valued words, where $n$ ranges from 1 to 6, inclusive. TF-IDF serves to extract the most representative words from each query [32]. Out of the 518 questions, 209 questions contained at least six words. For a fair comparison across the different lengths, only these 209 questions were considered - we call this the *Stack Overflow* dataset. We conducted the same automated evaluation pipeline as [18, 31], with the same threshold number.

We report the number of questions that *NCS* answered correctly in the top 1, 10 results in Figure 2. Looking at the trends for the solid lines (*NCS*), the number of questions answered correctly decreases as the length of the query decreases. With these observations, there is a clear need for a code search tool that is able to expand the query to help narrow down the intent.

Automatic query expansion is a natural approach to this problem. On a high level, a query expansion model would accept some keyword set $X_{\text{query}}$ and output another keyword set $X_{\text{exp}}$. If $X_{\text{query}}$ and $X_{\text{exp}}$ occur in similar contexts, then the combined query $X_{\text{query}} \cup X_{\text{exp}}$ would have better retrieval performance.

We have implemented our query expansion technique in a system called *NQE*. In this paper, we present an empirical evaluation of the method using a large corpus of Android applications. Our experiments clearly indicate the advantage of a query expansion approach over the existing NCS approach, as well as traditional feature-based code search, on "short" queries. The dashed lines in Figure 2 show the results for the same experiment using *NQE*. For all of the top-k results, performance across all query lengths increases. Table 1 shows concrete examples of where *NQE* improved the *NCS* rank on manually labeled queries.

**Contributions** Our contributions are the following:

- We define the problem of query expansion in code search, in a way that is driven by the corpus being searched, rather than being driven by past queries.
- We present a neural model, *NQE* that predicts the most promising keywords with which to expand a code search query. Experiments show that *NQE* can significantly improve the effectiveness of code search; in fact, we show this holds for two different code search methods: *NCS* and *BM25*.
- We show that *NQE* outperforms a baseline model for query expansion based on frequent itemset mining.

**Outline** The paper is organized as follows. Section 2 discusses related work. Sections 3 presents the main contribution of this paper, *NQE*. Section 4 describes the technical details for the existing models that were used for the paper. Section 5 presents the evaluations performed for the models. Section 6 provides discussion and results for the research questions. Finally, section 7 concludes the paper with possible future explorations.

## 2 Related Work

In this section, we discuss some related work in this area.

**Code Search.** There has been growing interest in the software engineering community [19, 20, 25] in going beyond grep-based tools. These methods utilize information about the code beyond lexical matching in order to retrieve and rank more relevant results. For instance, RCAS [16] uses graphs of relationships between JavaScript methods (e.g., sequencing, conditional, callbacks) to provide more relevant results to a natural language query from which a relationship graph is also inferred.

More recently, machine learning techniques have shown promise in addressing the inherent noise associated with searching in a code corpus. NCS [31] uses cosine distance between the neural embeddings of code snippets and the input query to compute relevance. Aroma [18] uses a nearest neighbor search but also supports searching based on richer features such as code structure, allowing it to be a "code recommender". CODEnn [8] jointly embeds code snippets and NL descriptions into a unified vector space such that a code and its description have similar vectors. In general, our work is applicable to most code search methods that take NL keywords as input. In our experiments, we evaluate it on NCS.

**Query Reformulation.** The fact that the results of a retrieval task depend largely on the quality of the query [13, 24] has prompted research into improving the latter. Conquer [13, 30] is a query reformulation tool that uses NL techniques to find co-occurring sets of keywords among query results to help a programmer refine the subsequent query. In our experiments, we compare with an implementation of this basic idea, but since our goal is query *expansion* rather than refinement (i.e., we do not have a notion of subsequent query), we extract co-occurring words in the corpus rather than query results. Lu et al. [17] use WordNet, a public thesaurus of English words, to lookup synonyms of query words and suggest them for expansion. Query reformulation has also been studied outside the context of code search, for example in bug localization [26].

In exploring machine learning methods for query reformulation, Refoqus [9] trains a classifier on a data set of past queries and relevant results in order to recommend a reformulation strategy that could improve results. Our work is fundamentally different from this, as we work with only the corpus of code for query expansion, rather than past searches. Imani et. al [14] also operate in the domain of neural query expansion, but train/evaluate on news-based datasets and address a slightly different problem of determining whether two expansions are from the same class, which they solve using Siamese networks. To the best of our knowledge, we are the first to explore machine learning based methods for query expansion in the context of code search.

**Deep Learning for Code.** The general area of using machine learning techniques to address problems in programming languages has seen a recent surge of interest. DeepCoder [4], PHOG [5], and Bayou [23] use deep learning models to synthesize programs from input specifications in the form of I/O examples or NL keywords. There is also significant work in using deep learning for predicting code properties and types [11, 27, 28], detecting bugs [22, 34], and representing programs in a vector space for various applications [1, 2, 12].

## 3 Neural Query Expansion

We now present the main contribution of this paper. First, we establish some definitions that we will use in presenting our models.

**Preliminaries.** Let $\mathcal{D}$ be a corpus of programs, where each program is considered as a "document" $d$. Let $V_m$ be the vocabulary of all method names in $\mathcal{D}$. Further, let us assume a function *split* that, given a method name, splits it into tokens based on CamelCase and snake_case. We can define $V_k$, a vocabulary of keywords, to be the set union of the result of splitting each method name in $V_m$, i.e., $V_k = \bigcup_{v \in V_m} split(v)$.

A *query* used for search is a set of keywords $X = \{x_1, \ldots, x_n\}$ where each $x_i \in V_k$. The *result* of a query is a ranked list of documents, represented as a sequence $R = \langle d_1, \ldots, d_k \rangle$ where each $d_i \in \mathcal{D}$. Each query $X$ is associated with a particular *expected document* $d_X \in \mathcal{D}$ that the user of a code search tool would expect it to retrieve. Given a search result $R$, the *rank* of the expected document is the index $i$ at which it appears in $R$, defined as $rank(d_X, R)$. For ease of definition, $R$ is always assumed to contain all documents in $\mathcal{D}$, and so a better search result is simply indicated by a numerically lower rank for $d_X$.

A *search tool* can be interpreted as a function $M : \mathcal{P}(V_k) \to S(\mathcal{D})$[1] that takes in a query $X$ and performs a retrieval operation to return a search result $R$. A *query expansion model* $Q : \mathcal{P}(V_k) \to \mathcal{P}(V_k)$ is a function that takes as input a query $X$ and produces another set of keywords $X_{\text{exp}}$, such that the expanded query including the keywords in $X_{\text{exp}}$ would result in a better search result. In other words, the goal of a query expansion model is to attempt to ensure:

$$rank(d_X, M(X \cup X_{\text{exp}})) < rank(d_X, M(X))$$

**Model.** *NQE* is a query expansion model where the function $Q$ is realized as neural network, as shown in Figure 3. It is an encoder-decoder model that, given a query $X$ as input, first produces a sequence of method names $Y = \langle y_1, \ldots, y_m \rangle$ where each $y_i \in V_m$, from which the final output $X_{\text{exp}}$ is obtained using the *split* function. The reason for using this two step process instead of directly generating the set of keywords $X_{\text{exp}}$ is explored further in Section 7.

First, given an input query $X = \{x_1, \ldots, x_n\}$, each $x_i$ is converted into an *embedding* $e(x_i) \in \mathbb{R}^g$, where $e$ is the embedding function learned during training and $g$ is the embedding dimension. The encoding of the entire query $X$ is then computed as the sum of the embeddings of each $x_i$, represented as $e_X = \sum_i e(x_i)$.

The encoding is then fed as the initial state to a Recurrent Neural Network (RNN) decoder that uses Gated Recurrent Units (GRU). The decoder's recurrent step function computes the probability distribution over the next output method name conditioned on previously generated names and the encoding of the input query[2], i.e., $P(y_{t+1}|y_1, \ldots, y_t, e_X)$ at time step $t$. The decoder repeatedly applies the step function and samples an output from the resulting distribution, until a special "end-of-sequence" token is sampled.

[1] $\mathcal{P}$ refers to the power set and $S$ refers to the set of all permutations

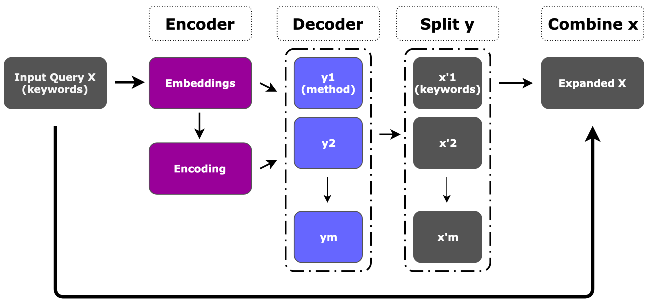

**Figure 3.** *NQE* Pipeline.

The final sequence of method names $Y = \langle y_1, \ldots, y_m \rangle$ is obtained by collecting the output at each time step. Finally, the output set of expanded keywords is obtained by splitting each method name in $Y$, i.e., $X_{\text{exp}} = \bigcup_i^m split(y_i)$.

The crux of why *NQE* is able to find co-occurring keywords is that the conditional distribution $P(y_{t+1}|y_1, \ldots, y_t, e_X)$ learns to predict method names that contain the query keywords and co-occur with each other in the underlying corpus.

On top of this core model, there are several enhancements we apply which we will cover only in brief. First, instead of obtaining a single sequence $Y$ from the decoder, we use *beam search* to obtain the top-$k$ most likely sequences of method names. Essentially, with beam search the decoder represents a distribution $P(Y|X)$ from which we can obtain the most likely sets of expanded keywords using the process described above.

Second, the decoder also contains a learned *attention* mechanism that combines the GRU's hidden state with the embedding set when determining what token to produce. The purpose of attention is to dynamically identify instances where certain parts of the input query are "relevant" to the next token that should be produced. Our model uses the attention algorithm proposed in [3].

For our experiments, we set both the embedding size and the hidden dimension of the GRU to 256. We set the learning rate to 3e−5 and the dropout to 0.4. Finally, we use the Adam optimizer [15] for training.

## 4 Background for Evaluation

We now setup a framework on which we evaluate *NQE*. The goal is to compute $rank(d_X, M(X \cup X_{\text{exp}}))$ and $rank(d_X, M(X))$ for some benchmark queries $X$, and assess whether the former is lower than the latter. For this, we evaluate *NQE* on two instantiations of $M$, the search tool: *NCS* and *BM25*. We also evaluate an alternate query expansion model for producing

[2] Internally, an RNN decoder would also utilize a hidden state but we omit it here for brevity, as it is fairly standard.

$X_{\text{exp}}$, using frequent itemsets. Here we provide an overview of these methods.

### 4.1 Search Tools

As defined before, a search tool is a function that accepts a query $X$ and returns a search result $R$. Here, we explore two possible instantiations of search tools: one based on neural embeddings (*NCS*) and another based on traditional information retrieval (*BM25*).[3]

#### 4.1.1 NCS

*NCS* is a novel code search tool introduced in [31]. The model exploits the concept of embeddings to represent both the query and documents (code snippets) as vector representations. *NCS* is an unsupervised model, as the embeddings are trained directly on the code corpus, such that words that appear in similar contexts within the corpus are close together in the vector space. This is based on the distributional hypothesis from NLP [10].

For each method body in the code corpus, certain tokens (e.g. method calls, comments, class names, etc.) are extracted and tokenized. Using fastText [6], *NCS* learns an embedding matrix $T \in \mathbb{R}^{|V_k| \times g}$, where $V_k$ is the token vocabulary, $g$ is the embedding dimension, and the $i^{\text{th}}$ row in $T$ is the vector representation for the $i^{\text{th}}$ word in $V_k$.

With the embedding matrix, *NCS* creates document embeddings for the code corpus by taking the weighted average of embeddings for the set of tokens in the document, where the weights are derived using TF-IDF [32]. With this step, we have an index matrix $S \in \mathbb{R}^{|\mathcal{D}| \times g}$.

Given a query, *NCS* creates the query embedding by taking the average of the set of tokens in the query. Then, the documents are ranked by cosine similarity.

#### 4.1.2 *BM25*

We also evaluate with an alternate search tool that is similar to *NCS*, but instead of embeddings, it uses *BM25* [21] to rank the documents. *BM25* is a well-known information retrieval technique, similar to the computation for TF-IDF. It uses the following formula to give score for a given document $d$ and a query word $x$.

$$\text{BM25}(d, x) = \text{IDF}(x) \cdot \frac{\text{TF}(x, d) \cdot (k + 1)}{\text{TF}(x, d) + k \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})} \quad (1)$$

TF is the term frequency, IDF is the inverse document frequency, $|d|$ is the length of the document $d$, and *avgdl* is the average document length of all $d \in \mathcal{D}$. $k$ and $b$ are tunable parameters.

[3] There exist other code search tools that may perform better. Our goal, however, was to not to find the best code search tool; rather, we wanted to explore the effect of a tool given varying query lengths, and whether query expansion could improve the tool.

In a search tool that uses *BM25*, the words are tokenized from each document in the same manner as *NCS*. The document vector, however, is calculated differently. It is a sparse vector of size $|V_k|$, the size of the vocabulary corpus. For each document vector, the $i^{\text{th}}$ entry is the *BM25* value for the word $x_i$ if that word is present in the document and 0 otherwise.

When a query comes in, the tool creates a multi-hot encoding query vector of size $|V_k|$, where the $i^{\text{th}}$ entry is 1 if $x_i$ is present in the query and 0 otherwise. Then, the documents are ranked by a simple dot product between the query vector and each of the document vectors.

### 4.2 Query Expansion Models

In addition to *NQE*, we evaluate a non-neural method for query expansion, namely *Frequent Itemset Mining*.

#### 4.2.1 Frequent Itemset Mining

We compare *NQE* with a non-neural model that captures the notions of co-occurring methods. To achieve this, we used Frequent Itemset Mining (*FIM*), a well-known technique for finding frequently appearing items in a dataset. In particular, we deployed the Apriori algorithm [33].

The underlying idea of *FIM* is that if a certain set of items appears frequently together, they are more associated with each other. Given a set $s_b$ of $k$ items, the Apriori algorithm attempts to find an extension $s_e$ that makes the set $s_b \cup s_e$ (size $k + 1$) most likely to appear in the dataset. This metric is calculated by the *confidence* of a particular expansion $s_e$ given a set $s_b$, where *support* is the number of times a set appears together in a dataset.

$$\textit{confidence}(s_b \longrightarrow s_e) = \frac{\textit{support}(s_b \cup s_e)}{\textit{support}(s_b)} \quad (2)$$

For our model, we set the minimum support to be 3, the minimum confidence score to be 0.5, and calculate frequent itemsets up to 4 items. [4] Given a query, for each combination set of three keywords, $s_b$, and for each possible expansion $s_e \in V_k$, we rank by the *confidence*$(s_b \longrightarrow s_e)$. If there are no expansions, then the same process occurs using two keywords, and then one. We return only one possible expansion, for a conservative expansion. [5]

## 5 Evaluation

### 5.1 Data Collection

The models are trained on a corpus of 737 public Android repositories cloned from GitHub. There are a total of 105,747 files, from which we are able to scrape 308,309 valid method bodies ($|\mathcal{D}|$).

[4] Although the maximum length of input queries is 6, the potential performance increase of calculating frequent itemsets for more than 4 items did not outweigh the computation and memory consumption.

[5] We tried varying number of expansions from 1-4, and found that one expansion yielded the best results.

The word embeddings for *NCS* are trained on this entire dataset. For *NQE*, we divide the dataset into 95%, 3%, and 2% for training, testing, and validation, respectively. Evaluation for all the models is carried out on the testing dataset. To address the problem of a large vocabulary size, we limit the the vocabulary corpus by filtering method names that appear in fewer than 3 method bodies and replacing them with an <*unk*> token. After this filtering, the vocabulary size of keywords $|V_k|$ is 6,896, and the vocabulary size of method names $|V_m|$ is 44,279.

## 5.2 Generating $X_{\text{query}}$ from $Y$ (*TF-IDF* dataset)

From the dataset, we extract $X_{\text{query}}$ (keyword set) and $Y$ (method sequence) from each method body $d$ such that the tokens of $X_{\text{query}}$ represent $Y$. To explore the effects of varying query length on the performance of the models, several samples of $X_{\text{query}}$ are chosen from $Y$. First, from $d$, the method calls are extracted to form $Y$. Then we take the top 50% TF-IDF methods from $Y$ to keep the most representative method calls. From here, *keywords* are extracted by tokenizing the method calls in the same manner as *NCS* (split by snake and camel case, filter out stop words, etc). There are two ways to form candidate tokens ($X_{\text{cand}}$) from $Y$: 1) Take the top 75% TF-IDF keywords from all of the keywords combined from $Y$, and 2) Get the top 1 IDF keyword from each $Y$. Since the first method chooses keywords from all of the combined keywords from $Y$, there is a possibility of localizing the intent to a particular subset of $Y$. The second method serves to provide a more broader intent for the entire $Y$. From $X_{\text{cand}}$, $X_{\text{query}}$ is sampled where $|X_{\text{query}}| \sim \text{Uniform}(1, 6)$. After sampling, we have approximately 1.8 million $(X_{\text{query}}, y, d)$ datapoints.

## 5.3 Manually creating $X_{\text{query}}$ (*Manual* Dataset)

We apply TF-IDF in Section 5.2 because we believe that TF-IDF serves to extract the most representative and human-like queries. To validate this, we also asked developers to perform the following task: given $Y$, create an $X_{\text{query}}$ from lengths 1 to 6. Our expectation is that if TF-IDF provides human-like queries, then the results between the two datasets should be similar. We obtained queries for 140 method bodies.

## 5.4 Evaluation Pipeline

In this paper, we use three different evaluation datasets:

- *TF-IDF* dataset: described in Section 5.2
- *Manual* dataset: described in Section 5.3
- *Stack Overflow* dataset: described in Section 1

We then compare three evaluation pipelines for code search. Given $X$, $d$, *SearchTool* $\in$ {*NCS*, *BM25*}, and *Expansion* $\in$ {*NQE*, *FIM*}, the model pipeline is as follows:

- **SearchTool**: $X$ is used as input to $SearchTool$.
- **Expansion + SearchTool**: $X$ is used as input to *Expansion*, which predicts $X_{\text{exp}}$. $X \cup X_{\text{exp}}$ is used as input to $SearchTool$.

## 5.5 Evaluation Metrics

We use this dataset of $X$ and $d$ to compare our models with two experiments.

First, we use the *TF-IDF* and *Manual* datasets to evaluate whether the model can retrieve the correct $d$ in the top 1, 10 results, along with the Mean Reciprocal Rank (MRR).

The second experiment uses the *Stack Overflow* dataset. Given a subset of a Stack Overflow question, an automated evaluation pipeline [18] determines whether the model retrieves a code snippet $d$ that is similar to the accepted Stack Overflow answer. Using an automated evaluation pipeline is crucial because manually assessing each result is difficult to scale. This pipeline uses the same similarity threshold as used in [31]. We report whether the model can retrieve a correct code snippet in the top 1, 5, 10 results.

# 6 Results

**RQ1: Does NQE improve performance for shorter queries?** Our experiments demonstrate that *NQE* improves the performance on shorter queries in all three datasets using *NCS SearchTool*.

*Stack Overflow*: From the top half of Table 2, we observe that *NCS +NQE* generally outperforms *NCS* across shorter query lengths. For example, in Top 5 retrieval results for queries of length 1, *NCS* retrieves a relevant result for 15 questions, whereas *NCS +NQE* retrieves a relevant result for 22 questions. When all of the query words are used as the initial input, *NCS +NQE* matches *NCS*, as shown in the last three columns. *NCS +NQE* also outperforms *NCS+FIM* across all query lengths, as shown in the bottom half of Table 2.

*TF-IDF & Manual*: The left halves of Table 3 and Table 4 show that *NQE* improves the *NCS* MRR score for 1 word and 2 word queries in both the *TF-IDF* and *Manual* datasets. The reason for this can be explained by the left halves of Figure 4 and Figure 5. In both datasets, *NQE* improves Top 1 accuracy for 1 and 2 word queries, and it also increases the Top 10 accuracy for 1 word queries. Conversely, for 3 word queries and longer, *NQE* worsens the *NCS* MRR, which is also explained by the decreased Top 1 and Top 10 accuracies. Figure 6 provides a deeper dive into this, where we measure the rank change on a per-sample basis. The figure shows that for Top 1 results, *NQE* improves (green lines) the *NCS* ranking more often than making it worse (red line) for queries of length 1 and 2. As the query length increases, both *NCS* and *NCS+NQE* tend to find the result in the first result (cyan line in the left plot).

Table 4 and Figure 5 show that *NQE+NCS* outperforms *FIM + NCS* on the *Manual* dataset, and the same generally holds true for the *TF-IDF* dataset with the exception of the bottom left plot of Figure 4. The similar trends between the two datasets corroborate our previous claim that the *TF-IDF* dataset is fairly representative of how a developer would search given a limited query length.

**Table 2.** The number of Stack Overflow questions answered in the top 1, 5, 10 results with varying lengths of the queries. Search performance increases when *NCS* is aided by *NQE*, especially for shorter queries.

| Top K | Query Length | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | 4 | | | All | | |
| | NCS | NCS+ FIM | NCS+ NQE | NCS | NCS+ FIM | NCS+ NQE | NCS | NCS+ FIM | NCS+ NQE |
| 1 | 10 | 10 | **14** | 12 | 12 | **16** | 20 | 18 | **22** |
| 5 | 15 | 15 | **22** | 20 | 21 | **29** | 33 | 28 | **34** |
| 10 | 18 | 18 | **25** | 23 | 25 | **33** | 40 | 40 | 40 |
| | BM25 | BM25+ FIM | BM25+ NQE | BM25 | BM25+ FIM | BM25+ NQE | BM25 | BM25+ FIM | BM25+ NQE |
| 1 | 11 | 11 | **17** | 13 | 14 | **16** | 16 | 16 | **18** |
| 5 | 17 | 15 | **20** | 21 | 20 | **22** | **26** | 22 | 24 |
| 10 | 19 | 17 | **24** | 22 | 21 | 22 | **30** | 27 | 28 |

**Table 3.** MRR results on *TF-IDF* dataset. Note that *NCS* + *NQE* outperforms *NCS* on short queries of length 1 and 2.

| Query Length | Mean Reciprocal Rank | | | | | |
|---|---|---|---|---|---|---|
| | NCS | NCS + FIM | NCS + NQE | BM25 | BM25 + FIM | BM25 + NQE |
| 1 | 0.092 | 0.109 | **0.284** | 0.060 | 0.045 | **0.219** |
| 2 | 0.416 | 0.428 | **0.543** | 0.276 | 0.193 | **0.390** |
| 3 | **0.672** | 0.547 | 0.574 | **0.528** | 0.356 | 0.424 |
| 4 | **0.807** | 0.706 | 0.650 | **0.657** | 0.494 | 0.542 |
| 5 | **0.852** | 0.727 | 0.679 | **0.649** | 0.491 | 0.531 |
| 6 | **0.951** | 0.839 | 0.812 | **0.729** | 0.574 | 0.605 |

**Table 4.** MRR results on *Manual* dataset. Note similar trends to Table 3.

| Query Length | Mean Reciprocal Rank | | | | | |
|---|---|---|---|---|---|---|
| | NCS | NCS + FIM | NCS + NQE | BM25 | BM25 + FIM | BM25 + NQE |
| 1 | 0.040 | 0.080 | **0.178** | 0.035 | 0.049 | **0.139** |
| 2 | 0.319 | 0.292 | **0.352** | 0.272 | 0.258 | **0.310** |
| 3 | **0.545** | 0.381 | 0.456 | **0.440** | 0.310 | 0.396 |
| 4 | **0.706** | 0.438 | 0.560 | **0.573** | 0.364 | 0.492 |
| 5 | **0.782** | 0.430 | 0.609 | **0.690** | 0.378 | 0.547 |
| 6 | **0.814** | 0.481 | 0.626 | **0.721** | 0.452 | 0.589 |

**RQ2: How does the quality of NQE sequence prediction affect the end-result?** Our experiments demonstrate that the sequence prediction quality impacts the *NCS+NQE* performance. On the *TF-IDF* dataset, *NQE* predicted the correct sequence in 30.3% of cases. Further, we observe that in 93% of the cases, at least 50% of the query appears within the keywords of the tokenized prediction, which supports our claim that *NQE* learns to predict methods relevant to the query. We also observe that in 100% of the test dataset, at least 50% of the predicted methods co-occurred in the underlying dataset, which suggests *NQE* understands the notion of co-occurring methods.

Figure 7 provides a breakdown across three different situations: a correct prediction, an incorrect prediction, and a "close" prediction. The first two are self-explanatory. A "close" prediction is defined as the Jaccard similarity between the method-set of the incorrect prediction and the method-set of the expected sequence being equal or greater than 0.5. Note that it is possible for an incorrect prediction to have a Jaccard similarity of 1 if, for example, it is a permutation of the sequence. The effect of a correct prediction is as expected; at low query lengths, it is more likely to improve the *NCS* rank, and as the query length increases, this scenario gradually becomes such that the *NCS* retrieval is already perfect and *NQE* cannot improve it. This is depicted by the inverse relationship between the green and cyan lines. For an incorrect prediction, *NQE* is more likely to worsen the rank than improve it, as shown by the red line consistently above the green line. When the prediction is "close", we see an interpolation between the two previous plots. *NQE* clearly improves the *NCS* rank more often than worsening it at short query lengths, but for lengths $\geq 3$, there are no clear differences in the two outcomes, as shown by the green and red lines closely following each other.

**RQ3: Do these findings generalize to other search techniques?** Our findings indicate that the results generalize to *BM25* in all three datasets.

From the bottom half of Table 2, we observe that *NQE* + *BM25* retrieves more successful results than *BM25* alone on the *StackOverflow* dataset. The right halves of Table 3 and Table 4 show that *NQE* improves the *BM25* MRR score for 1-word and 2-word queries in both the *TF-IDF* and *Manual* datasets. The right halves of Figure 4 and Figure 5 show that in both datasets, *NQE* improves Top 1 accuracy for 1 and 2 word queries, and it also increases the Top 10 accuracy for 1 word queries.

## 7 Future Work

One modification to *NQE* we hope to explore is sequence prediction at the sub-token level. This would afford us two advantages: a smaller decoder vocabulary size and the ability to handle rare method names that contain common sub-tokens. However, there is a trade-off between the expressiveness of the decoder and the accuracy of the sequence prediction that we would then need to consider. This is because if the internal RNN trains over sub-token sequences, it would generally train over sequences of longer and more varied lengths. Variance in length typically causes RNN performance to regress.

*NQE* internally predicts a sequence for the end goal of set expansion. Our rationale for using a sequence is that there currently do not appear to be any widely accepted neural models for set expansion. Our results show success for the reasons discussed in Section 3; *NQE* predicts method names that contain the query keywords and has a notion of co-occurring methods. Even then, an avenue of exploration for us would be to compare *NQE* with the proposed set expansion methods in [35] or [7]. Classifier chain methods exist [29], but on a high level the training data required for a set expansion task is exponential in size compared to the data needed for RNN-based models. Another possibility for us would be to explore methods that leverage the sequential

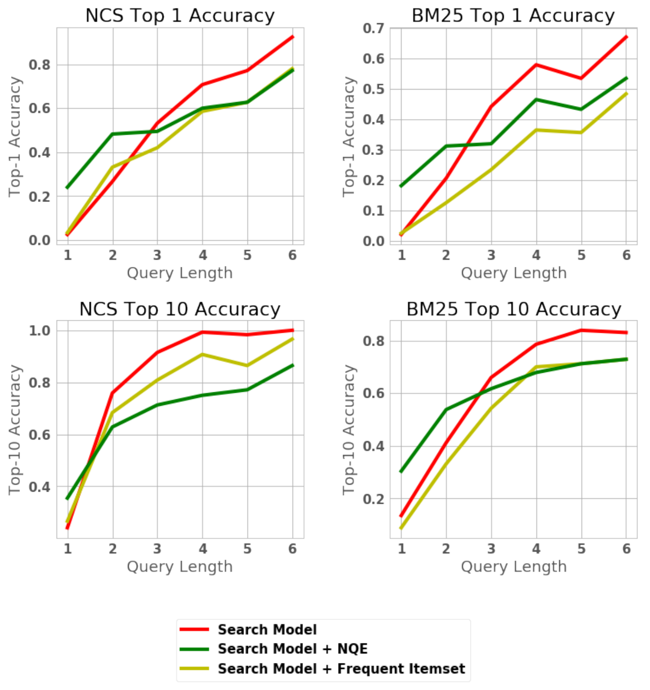

**Figure 4.** Top K accuracy on the *TF-IDF* dataset between *NCS* and *BM25* with and without *NQE* and *FIM*.

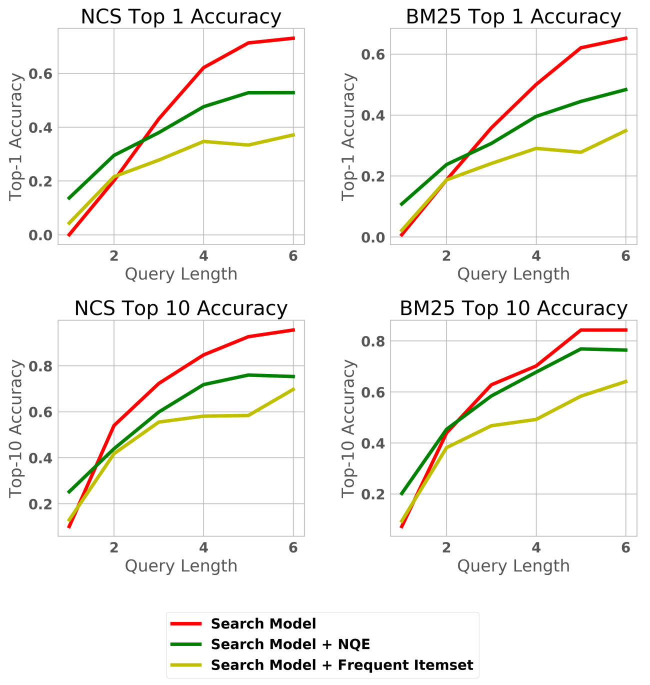

**Figure 5.** Top K accuracy on the *Manual* dataset between *NCS* and *BM25* with and without *NQE* and *FIM*.

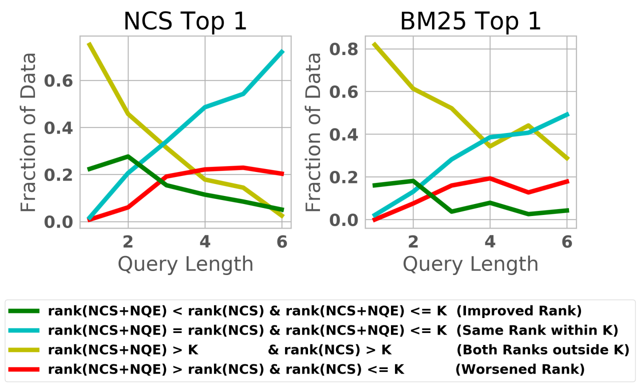

**Figure 6.** Top 1 ranking changes between *NCS+NQE* vs *NCS*.

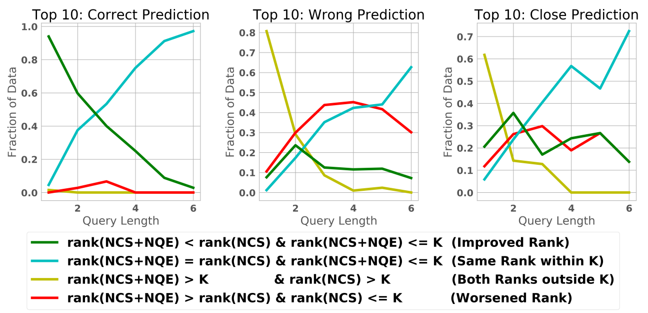

**Figure 7.** Top 10 ranking changes between *NQE* + *NCS* vs *NCS* when the prediction is correct, wrong, or close on the *TF-IDF* dataset.

information of the *NQE* output. Beyond method sequences, we may also look into producing syntactic information that could also be used in the search query.

## 8 Conclusion

In this paper we explored the performance of code search on varying query lengths. We found that *NCS* has a more difficult time retrieving the correct code snippet with shorter queries. Furthermore, developers' code search history logs show that shorter queries have more query reformulation and browsing time spent compared to longer queries. These observations lead us to believe that a model to expand the original query will be helpful to the developers.

We present *NQE*, an neural model that takes in a set of keywords and predicts a set of keywords, which then can be used to expand the query to *NCS*. Our results show that with query expansion, *NQE* + *NCS* is able to perform better than simply using *NCS*.

## References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM*

*on Programming Languages*, 3(POPL):40, 2019.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[7] Tian Gao, Jie Chen, Vijil Chenthamarakshan, and Michael Witbrock. A sequential set generation method for predicting set-valued outputs. *arXiv preprint arXiv:1903.05153*, 2019.

[8] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[9] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[10] Zellig S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954.

[11] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 152–162. ACM, 2018.

[12] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174. ACM, 2018.

[13] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. Nl-based query refinement and contextualized code search results: A user study. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 34–43. IEEE, 2014.

[14] Ayyoob Imani, Amir Vakili, Ali Montazer, and Azadeh Shakery. Deep neural networks for query expansion using word embeddings. In *European Conference on Information Retrieval*, pages 203–210. Springer, 2019.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware code search for javascript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 690–701. ACM, 2016.

[17] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549. IEEE, 2015.

[18] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *CoRR*, abs/1812.01158, 2018.

[19] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[20] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM, 2011.

[21] Bhaskar Mitra and Nick Craswell. Neural models for information retrieval. *CoRR*, abs/1705.01509, 2017.

[22] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 151–162. ACM, 2017.

[23] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[24] Nina Phan, Peter Bailey, and Ross Wilkinson. Understanding the relationship of information need specificity to search query length. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 709–710. ACM, 2007.

[25] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[26] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 621–632. ACM, 2018.

[27] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[28] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[29] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 254–269. Springer, 2009.

[30] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails. Conquer: A tool for nl-based query refinement and contextualizing code search results. In *2013 IEEE International Conference on Software Maintenance*, pages 512–515, Sep. 2013.

[31] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41. ACM, 2018.

[32] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[33] Hannu Toivonen. *Apriori Algorithm*, pages 39–40. Springer US, Boston, MA, 2010.

[34] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719. ACM, 2016.

[35] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.