Retrieval on Source Code: A Neural Code Search

Saksham Sachdev
University of Waterloo
Canada
me@saksham.me

Hongyu Li
Facebook, Inc.
U.S.A.
hongyul@fb.com

Sifei Luan
Facebook, Inc.
U.S.A.
lsf@fb.com

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

Koushik Sen
University of California, Berkeley
U.S.A.
ksen@berkeley.edu

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

Abstract

Searching over large code corpora can be a powerful productivity tool for both beginner and experienced developers because it helps them quickly find examples of code related to their intent. Code search becomes even more attractive if developers could express their intent in natural language, similar to the interaction that Stack Overflow supports.

In this paper, we investigate the use of natural language processing and information retrieval techniques to carry out natural language search *directly* over source code, i.e. *without* having a curated Q&A forum such as Stack Overflow at hand.

Our experiments using a benchmark suite derived from Stack Overflow and GitHub repositories show promising results. We find that while a basic word–embedding based search procedure works acceptably, better results can be obtained by adding a layer of supervision, as well as by a customized ranking strategy.

CCS Concepts • **Software and its engineering** → *Software development techniques*; *Software post-development issues*;

Keywords code search, word–embedding, TF–IDF

ACM Reference Format:
Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3211346.3211353

MAPL'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5834-7/18/06…$15.00
https://doi.org/10.1145/3211346.3211353

1 Introduction

The availability of immense — and rapidly growing — open–source code repositories have opened up the possibility of a new class of developer productivity tools that leverage statistical properties of code. The statistical knowledge embedded in these repositories has recently been harnessed for powerful tools for code completion, anomaly checking, type analysis, code synthesis, and so on.

In this paper, we focus on another kind of developer productivity tool: code search. Developers often search for examples of how to accomplish a certain coding task. Code search is a legitimate productivity booster, as it is much more efficient than working from the original documentation of the various APIs; the developer may not even know exactly which APIs to look for.

This is the reason Stack Overflow is so popular. While it caters to beginner programmers who may have questions on a programming language's syntax or semantics, a lot of usage of Stack Overflow is about looking for example usage of APIs. For example, an Android developer might want to quickly look up how to programmatically close or hide the Android soft keyboard, and indeed, the very same question has been asked and answered before on Stack Overflow[1].

However, as resourceful as Stack Overflow is, it does not contain an answer to every question. Additionally, questions specific to code and APIs proprietary to a company are not discussed in Stack Overflow, and comparable alternate Q&A forums may not be available.

Our premise is that with the availability of very large codebases, code fragments *related* to a developer's query are likely to already exist somewhere in those codebases. The challenge is whether these related code fragments can be retrieved without the availability of a curated Q&A forum. Searching over a curated Q&A corpus such as Stack Overflow is easy with well–known search techniques, but ability to search over raw, unannotated source code is not obvious.

[1] https://stackoverflow.com/questions/1109022/close-hide-the-android-soft-keyboard

To this end, we have been investigating how well basic natural language processing and information retrieval techniques applied to source code text directly can result in an effective code search system.

In this paper, we give a status report on our effort to build a search tool that works for large codebases — which we call NCS (for Neural Code Search) — that accepts natural language queries and shows related code fragments retrieved directly from code corpus. Our contributions are the following:

- We introduce a technique for building code search using a combination of word embedding [4], TF–IDF [14] weighting, and efficient higher–dimensional vector similarity search [9]. We discuss some of the relevant design tradeoffs involved in this process.
- We evaluate the effectiveness of NCS on open–source corpora, using a novel methodology that eliminates the need for a user study. Specifically, we collected a benchmark of 100 Android–specific queries from Stack Overflow, along with their correct code snippets from the post. As one of the criteria for selecting these questions, we ensured that those (or substantially similar) code snippets exist among the top 1000 Android projects on GitHub. We then measured how well NCS, when asked each of those Stack Overflow questions, was able to retrieve the corresponding code snippet from a clone of those GitHub repositories.
- Our results shows that this simple technique can work surprisingly well in the domain of source code. With our current model, we are able to answer almost 43% of the Stack Overflow questions we collected directly from code (including the question *"close/hide the Android soft keyboard"*.
- We discuss some of the limitations that we observed in the basic word–embedding based search. We propose two ways to mitigate the limitations: adding a layer of supervision, and a better, custom ranking of the results. These improvements further improve the search quality to be able to answer 68% of the questions correctly.
- Finally, we present a larger, automated evaluation on over 500 questions to compare the basic NCS, enhanced NCS as well as a couple of IR techniques: Elasticsearch and BM25.

The paper is organized in the following sections. In Section 2, we describe the NCS search model. Section 3 carries out a micro–benchmark–based evaluation, primarily for the purpose of evaluating design choices. Section 4 presents the Stack Overflow case study. Section 5 reflects the improvements to the search model based on the study results, and their effectiveness is tested on an automated evaluation framework, elaborated in Section 6. Finally, Section 7 presents related work.

2 Model

In this section we describe the details of NCS model. The main idea is to capture program semantics, or rather "intent", informally, using continuous vector representations [13]. Continuous vector representations, when computed appropriately, have the desirable property that vector representations of two semantically similar entities are close in terms of vector distance. We caution readers that these vector representations capture program intent in an informal way and are not designed to capture the mathematical meaning of a program.

In NCS, we create a continuous vector embedding of each code fragment at method–level granularity. Given a natural language query, we map the query to the same vector space. Then we use vector distance to simulate relevance of code fragments to a given query.

This overall process is fairly standard in NLP and document retrieval [17], but the details for code retrieval are sometimes significantly different, as we point out where relevant. We first describe how we extract "natural" information from source code, and then describe how we compute vector representations.

2.1 Extracting Information from Source Code

Our hypothesis is that tokens in source code contain enough natural language information to make retrieval possible. In this step we create a natural language "document" for a piece of code — at the granularity of a method or function — by extracting such information.

The simplest approach is to use a simple tokenizer to extract all words from source code by removing non–alphanumeric tokens. Being language–agnostic, this method is ideal for processing codebases composed of multiple programming languages, but on the other hand, it lacks the ability to leverage the syntax of any specific language; it is unable to differentiate whether a word comes from a variable name or a method call. We use a parser–based approach instead, in which we traverse through the parse tree for each method, and extract information from the following syntactic categories (here we assume a Java–like language, but one with optional types):

- Method name: We extract the method name and combine it with the class name in which the method is contained.
- Method invocations: Method invocations are arguably the most important information inside a code block because they give hints about what is happening in the code. Other features like control flow are also important but cannot be easily expressed in natural language.
- Enums: Unlike local variables, these constants are carefully named, containing meaningful information about the state of program.

- String literals: They usually include human–facing text and documentation, both containing useful words which we extract.
- Comments: They contain very useful information as they are already natural language descriptions of code.

We follow the common practice of *not* extracting variable names, as variable names would vary by developers, and moreover, their signal tends to be captured by surrounding method name already [15].

After extracting the aforementioned information, we process the source code words to transform the information to a natural language document. We split the camel–case (`camelCase`) or snake–case (`snake_case`) concatenated strings into separate lower–case words, remove non–alphanumeric characters, and filter out for various things including strings containing backslash (as they are usually escape characters), string longer than 300 characters, etc. Finally, we obtain a flat list of words extracted from source code that resemble a natural language document.

Figure 1 shows an example of the data preparation: the comments, method definitions, and method invocations from this example are scraped and cleaned for the model generation.

```
/**
  * Converts pixel in dp
  **/
public static int pxToDp(Context context, int px) {
    DisplayMetrics displayMetrics =
      context.getResources().getDisplayMetrics();
    return (int) ((px / dispMetrics.density) + 0.5);
}
```

```
converts pixel in dp px to dp get resources get
display metrics
```

Figure 1. Words extracted from a Java method [2]

2.2 Building Vector Representations

We first create continuous vector representations of words such that if the cosine distance between their vector representations is small, then those words tend to have related meanings (Section 2.2.1). We then use these word vectors to represent intents of source code as "document" vectors (Section 2.2.2), using which we retrieve results (Section 2.3).

2.2.1 Building Word Embeddings

Word embedding has become a popular concept since the Word2vec paper [13]. Word2vec computes vector representation of words using a two–layer dense neural network that can be trained unsupervised on a large corpus, such that words that share common contexts are located close together in the vector space. We use the converse of this statement to help define semantic relatedness: words with vectors that are closer together should have related meanings; this is called the distributional hypothesis in the NLP literature [7]. The Word2vec algorithm essentially produces a lossy encoding of words in a lower–dimensional vector space than the size of the vocabulary. With proper dimensionality and training, these embedding vectors retain the meaning of each word, in the sense described above.

Notice that word embedding models are originally designed for processing natural language corpus; our extracted corpus from source code is different from natural language text. Nevertheless, we rely on the assumption that the distributional hypothesis also holds for source code text, i.e. words that occur in the same context (in source code) also have related meanings. Table 1 shows examples of clustered word embeddings trained from our corpus, which supports this claim empirically. The work by Allamanis [2] also suggests that the hypothesis holds for source code.

In our implementation, we use a variant of Word2vec model, called FastText [4]. We employ the continuous skip–gram model with a window size of 5, i.e. all pairs of words within distance 5 are considered nearby words.

Table 1. Examples of word embedding synonyms trained from Java Android GitHub repositories

Word	Words with closest vector representations
button	*click, offlinepopup, buttons, dismissible, clicked*
keyboard	*softkey, softkeyboard, emojicons, soft, IME*

2.2.2 Building Document Embeddings

We express the intent of source code text in the same high–dimensional vector space as the word embeddings, by aggregating representation of all the words extracted from the source text as described in Section 2.1.

It turns out that the manner in which word embeddings are aggregated to obtain a document–level vector representation (or embedding) makes a significant difference. Whereas previous work [14] has suggested simply averaging word embeddings, we found this does not work well for our purposes.

We tried three variants of the combination method:

1. Average over all the words;
2. Average over the unique words in each document;
3. Weighted average of all unique words in a document according to Equation 1.

$$v_d = u\left(\sum_{w \in d} u(v_w) \cdot \text{tfidf}(w, d, C)\right) \tag{1}$$

[2] From https://github.com/sockeqwe/mosby (slightly abridged)

where d is a multiset of words representing a document; C is the corpus containing all documents; u is a normalizing function where $u(v) = \frac{v}{|v|}$. TF–IDF, short for term frequency–inverse document frequency, is a function that assigns a weight for a given word in a given document [14]. A word has a higher weight if it appears frequently in the document but is also penalized if it appears in too many documents in the corpus.

$$\text{tfidf}(w, d, C) = \frac{1 + \log \text{tf}(w, d)}{\log |C|/\text{df}(w, C)} \quad (2)$$

Experiments reported in Section 3.2.2 show that the weighted average method works significantly better than the other ones.

2.3 Retrieval

The search query is expressed as natural language sentences, such as "close/hide soft keyboard", "how to create a dialog without title". We simply average the vector representations of constituent words to create a document embedding for the query sentence; Words in query that never appeared in our code corpora are dropped from consideration.

We then use a standard similarity search algorithm to find the document vectors with closest cosine distance. FAISS is an implementation of various efficient similarity search algorithms, including a scalable nearest neighbor graph–based search [9]. We use this library in our implementation.

3 Evaluation

In this section, we evaluate the effectiveness of aforementioned approach at different design points. The purpose of this evaluation is to help us choose from several possible design points.

3.1 Evaluation Methodology

We use a *subset words retrieval metric* to evaluate search retrieval. The basic idea is that we take a subset of words from a given document to create (or rather, *simulate*) a query and feed the query to the system to see whether the given document can be retrieved given that subset of words. Some methods in the GitHub corpus are very short with not enough signal; for this reason, we limited these tests to methods that have at least five extracted words.

We have two kinds of "micro" benchmark tests to evaluate our system:

- Random Benchmark Test: use 20% randomly selected words in each document to create the query, with minimum of five words per query;
- TF–IDF Benchmark Test: use 20% highest TF–IDF valued words in each document to create the query, with minimum of five words per query.

Evaluation Metric: We evaluate the system by the percentage of documents that are retrieved back at first place, as well as within the top–9 places, using the subset words retrieval metric.

Evaluation Data: The evaluation data for the micro–benchmark is all 704,229 methods in the 1,000 Android repositories we cloned from GitHub. Table 2 shows promising results, as we were able to return the correct method in the top nine results for 74.4% and 94.6% of the queries for the random and TF–IDF benchmark tests, respectively. (It is expected for the random benchmark test to perform worse than the TF–IDF.)

Table 2. "Micro" benchmark test results for the two types of test. The document vectors were built with embedding dimension 500 with TF–IDF weighting.

Benchmark Test	Top 1	Within Top 9
Random	46.1%	74.4%
TF–IDF	68.9%	94.6%

3.2 Experiments

We report on three experiments using the preceding micro–benchmarking methodology. First, we investigate the impact of embedding dimension. Next, we compare three ways of combining word embeddings to document embeddings. Finally, we also compare the NCS pipeline with an alternative pipeline based on information retrieval alone, and that does not use embeddings.

3.2.1 Choosing Word Embedding Dimension

One challenge is to choose the optimal word embedding dimension size. We believed that the default dimension for fastText, 100, may be too small to capture the full intent of each vocabulary in our corpus. Thus, we experimented with four dimensions: 100, 250, 500, 750. The next table shows the results for the TF–IDF benchmark test, building the document vectors using TF–IDF weighting. The results, cited in Table 3, show that the benchmark tests for the higher dimensions performed better. This can be explained by our large corpus of documents and vocabulary. Because there was no significant difference between the results for 500 and 750 embedding size, we decided to use size of 500 for the model, as it optimizes both accuracy and speed.

3.2.2 Choosing Document Vector Building

To build the document vectors, we use TF–IDF weighting for all the unique words in the document. This approach was the result of experiment to compare several different methods to build the document vector. The three that we tried were:

- *Sum all words in the document*: this is the most primitive version, where we add all the word embedding vectors for each word in the document. This fails in the

Table 3. TF–IDF benchmark test results for different embedding dimensions. The document vectors were built with TF–IDF weighting.

Embedding Dimension	Top 1	Within Top 9
100	63.1%	91%
250	66.9%	93.6%
500	68.9%	94.6%
750	69.9%	94.6%

case where a method is heavily skewed by common or unimportant words.

- *Sum unique words in the document*: this method is to correct the problem stated in the previous method. Here, we only add the unique set of words in the document. While this solves the problem with methods containing a large number of unimportant words, another problem arises. If there are important, or "key" words that highlight the intent of the document, this method does not reflect the importance.
- *TF–IDF weighting*: this method aims to solve the issues presented by the methods above. This method takes the unique set of words in the document and multiplies each word by its corresponding TF–IDF score. This way, each word is accounted for by its importance.

Table 4 shows the results of this experiment. This model was built with embedding dimension 500 and tested on the TF–IDF benchmark test (see Section 3.1). It can be seen that the TF–IDF weighting outperforms the other two methods and is quite important for NCS effectiveness.

Table 4. TF–IDF benchmark test results for different ways of building the document vector. The document vectors were built with embedding dimension 500 with TF–IDF weighting.

Document Vector Method	Top 1	Within Top 9
Sum all	0.54	0.848
Sum over unique	0.518	0.822
TF–IDF weighted sum	0.689	0.946

3.2.3 Comparing against Information Retrieval

Using embeddings for retrieval can also be problematic because of the lossy representation, which loses the ability of supporting exact word matches. For this reason, we wanted to compare embeddings with a more traditional information retrieval technique.

We put together an alternate processing pipeline similar to NCS, but one that uses only a BM25–based [14] document similarity to serve queries. BM25 is similar to TF–IDF, with a slightly different score computation, as shown in Equation 3:

$$\text{BM25}(D,Q) = \sum_{q \in Q} \text{IDF}(q) \cdot \frac{\text{tf}(q,D) \cdot (k+1)}{\text{tf}(q,D) + k \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}. \quad (3)$$

where k, b are tunable parameters, $|D|$ is the length of a document, and avgdl is average document length.

In this pipeline, the information extraction from source code is the same as in NCS, but document vectors are computed in a different way, without using embeddings. We compute BM25 values for each word in each document for the entire corpus. The document vector is then just a sparse vector of these BM25 values for words that exist in the document, and zeros for words that do not. Query matching is done by a simple dot product between each document vector and a one–hot encoding of the query over the same vocabulary.

The evaluation result on the micro benchmark is displayed in Table 5. The performance of NCS is in the vicinity of BM25, which assuages our concern with the lossy representation used in NCS.

Table 5. TF–IDF benchmark test results for NCS vs BM25. The document vectors were built with embedding dimension 500 with TF–IDF weighting.

Search Method	Top 1	Within Top 9
NCS	68.9%	94.6%
BM25	73.6%	96.8%

4 Case Study: Stack Overflow Evaluation

We want to evaluate the effectiveness of Neural Code Search. In production, we can track developer behaviors on a search platform and IDEs to get click–through rates, but they are only proxy–metrics, and we do not know if they found what they were looking for.

We leverage Stack Overflow to work around this problem. Stack Overflow contains a large number of queries, as well as up voted answers that can be treated as the correct answer for that query, or in other words, the ground truth.

For this case study, we decided to measure the effectiveness of NCS on Android Java code, for which there are a lot of queries and answers on Stack Overflow, as well as a lot of large code repositories on GitHub. Given a Stack Overflow question as a query, NCS will retrieve a list of method from the corpus of Android projects from GitHub, and we consider a search successful if at least one of the methods in the top 10 search results match the Stack Overflow code snippet.

4.1 Data Preparation

To evaluate NCS, we need a set of Stack Overflow question and corresponding code snippet from the answers, and a

corpus of Android projects from GitHub in which the Stack Overflow code snippets exist. The latter is collected from the most popular 1,000 Android projects on GitHub.

We need to select the Stack Overflow questions and code snippets in an unbiased manner. To do this, we created a heuristics–based filtering pipeline where we discarded open–ended, discussion–style questions. We started with the top 17,000 frequently asked questions for Android on Stack Overflow and used numerous filtering techniques to reduce to 2,000 question and code snippet pairs.

Next, we need to verify these code snippets exist in the code corpus so that NCS can possibly retrieve them. This is done by searching the code snippets directly using Elasticsearch, with a custom tokenizer to split up camel case tokens. Since Elasticsearch is a fuzzy text search tool, we get many false positives in the results. We then manually chose 100 questions for which the correct code snippets exist in the GitHub corpus. This became our final data set for evaluating NCS.

We built a NCS model on the corpus of 1,000 Android projects from GitHub, and directly used the Stack Overflow question as the query to NCS. Then we manually evaluated each method retrieved by NCS to decide whether it is similar to the code snippet in the Stack Overflow answer, primarily based on whether it invokes the correct APIs and used the correct constants.

4.2 Results

The results are as follows:

- For 43 out of 100 questions, NCS found a similar or acceptable method to Stack Overflow that correctly answered the Stack Overflow question.
- For 8 out of 100 questions, NCS *almost* found a similar answer. These are the cases where the retrieved function calls another function that does the expected job.
- For the rest of the questions (49/100), NCS did not find a good enough answer. See analysis below.

The results show that NCS is able to answer succinct questions that have keywords similar to those that can be found in the function body. Here are some examples of queries NCS successfully answers correctly:

- *How to convert a Bitmap to Drawable in android?*
- *How to find MAC address of an Android device programmatically?*
- *How to handle back button in activity?*
- *Close/hide the Android Soft Keyboard.*

Interestingly, we found occasional (though not frequent) cases of synonyms coming in useful. For the question *How to turn on the GPS on Android*, the intended code fragment does not contain the word *GPS* but does contain the word *location*. NCS was able to retrieve it because of the use of

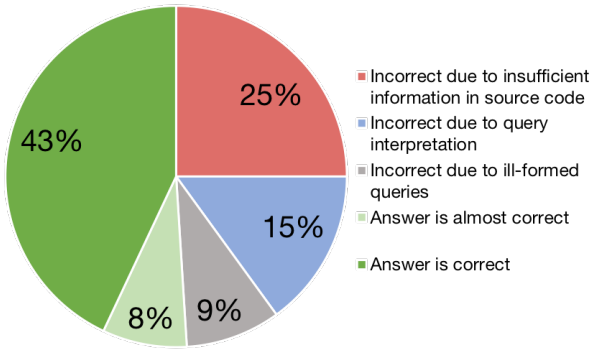

Figure 2. NCS evaluation results

embeddings. This is a potential advantage of the embeddings–based search, though on the whole BM25 worked comparably on this data set (47 questions answered correctly and 1 almost correctly).

There are three broad categories of reasons why NCS did not perform up to expectations:

- For 25 of the 49 failed questions, there were insufficient similarities between the information manifest in the source code and the words in the question. An example of this type of question is *"How to hide status bar in Android."* Here, hiding the status bar is akin to making the app full screen. However, NCS Is not aware of the association between full screen implying that the status bar is hidden; even word embeddings did not help in this case.
- For 15 of the 49 failed questions, there are query interpretation issues. NCS prioritized the wrong query words during search and reformulating the query would often lead to the correct answer. An example of this type of question in *"Remove underline from links in TextView."* Here, NCS finds great matches for the keywords *"underline"* and *"textview"* but not for *"remove"*. If we reformulate the query as, *"Underline textview"* or *"Delete underline textview,"* we get the right results.
- 9 of the 49 failed questions were either too complex (requiring multiple different files) or not well formulated. These are questions whose title does not necessarily represent the actual question being asked. The fourth question in the list below gives an example.

Here are some questions that NCS did not answer well:

- *How to hide status bar in Android?*
- *Remove underline from links in TextView.*
- *Android: How to create a Dialog without a title?*
- *How to get the result of* `OnPostExecute()` *to main activity because AsyncTask is a separate class?*

5 Improvements

After analyzing the questions which NCS did not answer well, we made two improvements to NCS that increase its performance significantly.

5.1 Query Enhancement

NCS assumes that the words in the query come from the same domain as the words in extracted from source code. As the next experiment shows, this is not always the case: From a dataset of 14,005 Stack Overflow posts, we took all pairs of questions and answers with positive scores, where the answer contains a code snippet. From this dataset of 235,989 natural language query and source code pairs, we extracted all unique words (for source code we use the same procedure as in Section 2.1). Figure 3 shows a Venn diagram of the two vocabularies from query domain and from source code domain. Out of 13,972 unique words in the queries, less than half (6,072 words) also exist in the source code domain. This means that if a query contains words that do not exist in source code, our model is not going to be effective in retrieving the correct method; such a word would have no presence in the embedding vector for the query. This observation motivates the need for a supervised learning to map words in the query into words in source code.

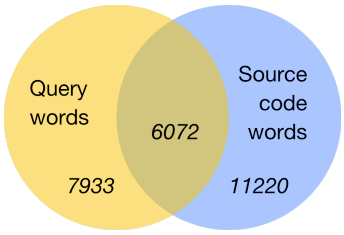

Figure 3. Overlap of words from queries and source code

We implemented a basic supervision to learn the translation from words in queries to words in code, using the dataset of 235,989 pairs of queries and code snippets. We lemmatize the query (natural language) words to remove variance due to inflectional endings; from the code snippets, we extract the method calls (by anything that has a left parenthesis immediately after it). The goal is to produce the highest co–occurring code word given a query word. Due to the noisy nature of Stack Overflow data, the initial step is to extract the most representative code words from each code snippet to obtain a cleaner dataset. To this end, we retain the highest 50% TF–IDF–scoring code words from each code snippet. Then we create a co–occurrence matrix C of size 14,005 by 17,292 (number of words in queries times the number of words in code), where C_{ij} is the count of Stack Overflow posts in which query word i and code word j appear together, with normalization. The normalizing factor ensures the row sum of this matrix is always 1.

The query is then enhanced in the following way: for each word w_i in the query, we pick the (code) word from matrix C with the highest co–occurrence value in row i, and only append that code word to the query if the value is significant enough ($C_{ij} \geq 0.05$). This threshold ensures that the code words added into the query are meaningful and highly–correlated with the original words. We can think of co–occurring words as a different kind of synonym. Table 6 shows examples of co–occurring words.

Table 6. Examples of co–occurring words from queries and from source code

Natural language word	Co–occurring code word
GPS	*location*
keyboard	*soft*
Internet	*network*
base64	*encode*
iterate	*next*

An example of co–occurring words helping retrieve the correct code snippet is the Stack Overflow query "*Android WebView: handling orientation changes.*" There is no overlap between the words in the query and words in the code, but the co–occurrence matrix added the word "orientation", which is correlated with "configuration". With the addition of this new word, NCS is now able to retrieve the correct method, which is named `onConfigurationChanged`.

We re–ran the evaluation in Section 4 with this query enhancement technique—adding co–occurring words as well as removing common stop words from the query—and found that NCS can correctly answer 55 questions (an increase of 12). We note that this method is not a complete translation from natural language query words to code words, since we are not replacing the natural language query words with source code words; we are simply adding words. In the future, we hope to train a single neural network using supervised learning to produce embeddings for words in both domains, such that natural language word embeddings are directly mapped into source code domain.

5.2 Ranking

We noticed during evaluating NCS results is that, while only 43 questions are correctly answered in the top 10 results, if we consider the top 20 results, NCS gets 59 questions correct. This inspired us to run an analysis on the search rank at which the first correct answer appears. The result is shown in Figure 4. (This data is obtained on a larger dataset of Stack Overflow questions, using the automated evaluation framework described in Section 6.)

From this distribution, we find more than half of the correct answers (224 out of 349 retrieved) within the top 500 rank are not in the top 10 results. This spread in the ranks is

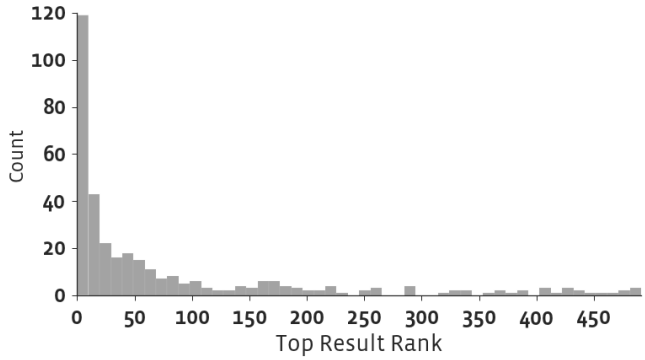

Figure 4. Histogram of ranks of first correct answers. There are only a few correct results beyond the top 500 rank so we decided to cut off at 500.

a limitation of embedding–based search algorithm, because continuous vector representations do not exactly capture the intent of either a query or a document.

Since most users would not browse beyond the first 10 results in a search session, this observation warrants a ranking strategy to bump up the correct answers to top. We first retrieve the initial 500 results with the highest cosine similarity, and then apply the following rules to rank the results:

- Up rank methods that are at least three lines long.
- Up rank methods containing at least 10 tokens.
- Up rank methods that match at least $n-1$ query words, where n is the number of words in the pre–processed query.
- Up rank methods that contain all n query words.
- Up rank methods that match at least $m-1$ original query tokens, where m is the number of tokens in the original query (without splitting).
- Up rank methods that contain all m query tokens.

The first two criteria prevent retrieving very short methods with no substance (in principle, we could have excluded these documents from consideration in the first place.) The rest of the rules reward those methods that most closely match the intent expressed in the query. Furthermore, these ranking rules are applied in the order of ascending importance. For example, it is more important for a method to contain all *unsplit* query tokens (perhaps a long camel–case class name), than that it contains all query words after pre–processing (perhaps scattered words).

We re–ran the automated evaluation after implementing the ranking heuristics on those questions that were originally in the top 500 rank, and found that the results improved significantly.

In Figure 5, the first box plot shows the same distribution as in Figure 4, where a right–skewed distribution indicates many correct answers are "buried" in the top 500 results. The second box plot shows the distribution of results after

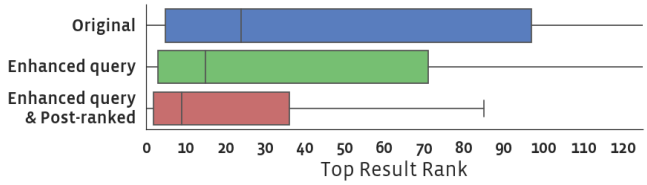

Figure 5. Comparison of distributions of ranks of first correct answers

we enhance the search query with co–occurring words. The result is better, but a long tail is still present. The last box plot shows the distribution of results after query enhancement and ranking heuristics. Now more than 50% of the retrievable results are in the top 10 range.

With these two additions, we re–evaluated NCS on the original 100 questions dataset using manual evaluation, and found that NCS can now correctly answer 68 questions, showing a 58% increase in accuracy.

6 Automated Evaluation

Manually evaluating NCS on these Stack Overflow questions does not scale. It is also difficult to measure iterative improvements to our search algorithm. Therefore, we developed a framework to automate the evaluation process using a similar code detection tool.

6.1 Evaluation Framework

We found that a large subset of correct code snippets retrieved by NCS contain (or *are*) almost the identical code snippet as in the Stack Overflow answer. Thus we developed a similar code detector that is capable of detecting near–clones of a given code snippet. In our case, we intend to find near–clones of the code snippet in the true answer from within a method body retrieved by NCS.

The automated evaluation pipeline works by first taking the top 10 results retrieved by NCS, and for each retrieved method, getting a similarity score between the ground–truth code snippet and the method. This detector automatically accommodates for the case that the ground–truth code snippet might be only a part of the retrieved method. If at least one of the top 10 NCS results contains a code fragment whose similarity score is above the threshold, we mark this question correctly answered by NCS. The similarity threshold we chose seeks to minimize false positives during evaluation. This approach does permit false negatives, but we get a lower bound on NCS retrieval without time–consuming manual analysis.

We first cross-checked the automated evaluation suite with our manual inspection on the original 100 questions; the result is shown in Figure 6. The automated similarity detection has a very low false positive rate, but has a non-trivial false negative rate. It tends to miss those methods

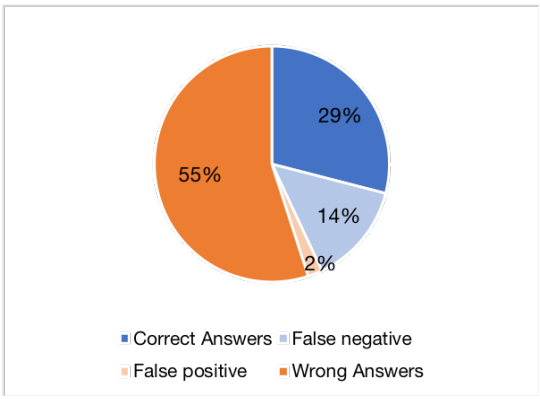

Figure 6. False positives and false negatives in automated evaluation

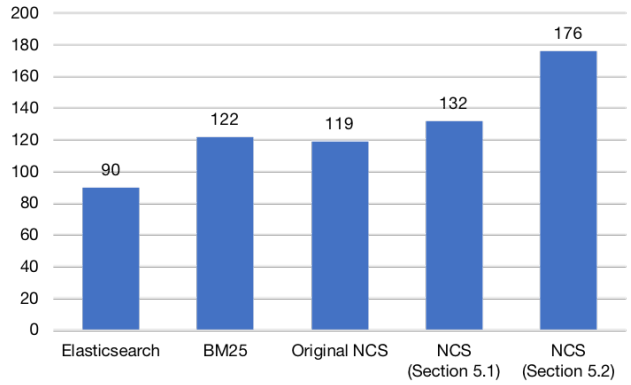

Figure 7. Comparison using automated evaluation on 518 questions

that invoke the same APIs but look different syntactically; in other words, the automated evaluation gives a lower bound on the actual search performance.

Using the similar code search tool we were also able to increase the size of our evaluation dataset in an automated fashion. Previously (see Section 4.1) we were not able to scale the filtering process because Elasticsearch is not intended for code search, and is therefore unreliable in determining whether a code snippet appears in the GitHub corpus. We replaced Elasticsearch with the similar code detector mentioned above, which is capable of finding a code snippet from a large corpus with high accuracy. This allows us to expand the evaluation dataset from 100 to 518 questions, on which we can apply the automated evaluation.

6.2 Comparison Results

Using the automated evaluation framework we compared the performance of NCS with various information retrieval techniques:

- Elasticsearch: a traditional information retrieval text search system. We tokenized the code from GitHub corpus and created an Elasticsearch index.
- BM25: we also built our own BM25 search using the same word splitting and preprocessing techniques.
- NCS: pure vector similarity search without the improvements described in Section 5.
- NCS (Section 5.1): in this version we enhance the query with co–occurring code words.
- NCS (Section 5.2): in this version we enhance the query *and* employ the ranking heuristics on the top 500 retrieved results.

Note that percentage-wise the numbers here (*e.g.* 176/518) look worse than those in Section 5 (*e.g.* 68/100), in part because the automation evaluation significantly under-reports the actual search performance. Yet, this is meaningful for comparing across techniques in a scalable way.

The results show that while the basic NCS achieves similar performance as the traditional information retrieval systems, by exploiting the mapping between query (natural language) vocabulary and source code vocabulary, together with ranking heuristics, NCS is able to retrieve significant number of additional correct code snippets than Elasticsearch and BM25.

7 Related Work

In the interest of space, we focus only on the most closely related work in code search.

SNIFF [6] is a code search engine that works by inlining API documentation in client code. SNIFF then indexes the annotated code for the purpose of code search. SNIFF also intersects and ranks search results to recommend most relevant and concise code snippets. SNIFF uses conventional indexing techniques to perform code search. Moreover, SNIFF depends on the availability of comments in the user code.

Given a natural language query, Portfolio [12] retrieves and visualizes functions and their usage chains that performs the task specified in the query. The motivation behind Portfolio is based on the observation that programmers are often more interested in finding definitions and usage of functions instead of relevant code snippets.

CodeHow [10] is a recent code search technique that first uses code documentation to retrieve possible API calls that could be use in the target code. CodeHow then augments the query with the retrieved APIs and performs search using the augmented query. This results in more accurate search results.

Sourcerer [3] is a framework for performing code search over open–source projects available on the Internet. Sourcerer works by extracting fine–grained structural features from source code. Keywords and features extracted from source code are then searched using the text search engine Lucene.

Chan et al. [5] proposed an approach where code can be searched using short text phrases as query. Their system

returns code represented as a graph whose nodes represent classes or methods matching the query keywords and whose edges denote the invocation relationship between the nodes.

SWIM [16] is a code synthesis technique which goes beyond simple code search. SWIM converts user queries into relevant APIs using click–through data from the Bing search engine. It then synthesizes code snippets containing these APIs.

Several recent work [2, 11, 15, 18] have shown how to *generate* code snippets given a natural language query. Allamanis et al. [2] uses probabilistic models to jointly model short natural language queries and code snippets. They show how to use the models to synthesize[3] a program snippet from a natural language query. Their technique requires training data that contains code snippets and their corresponding natural language descriptions. Murali et al. [15] proposes a deep–neural network based technique to synthesize a target program given a label specifying a small amount of information about the target program. For the purpose of training, their technique automatically extracts labels from code snippets. Their technique uses TreeLSTM to generate programs. Seq2SQL [18] uses a deep–neural network model to synthesize SQL queries from natural language questions. All of these technique create new code snippets given a query instead of returning a code snippet from an existing code repository.

Recently a number of models [1, 8] have been proposed to create high–quality natural language *summaries* of code snippets. CODE-NN [8] uses Long Short Term Memory (LSTM) networks with attention to produce sentences that describe C# code snippets and SQL programs. Allamanis et al. [1] uses convolutional–neural networks (CNNs) to summarize code snippets. Such techniques can be used for code search as follows. One first creates summaries of the method bodies of all the methods present in a code repository. The summaries are indexed and searched given a natural language query. Alternatively, using the trained model one computes the probability of the query sentence for each code snippet in the corpus and returns the snippet with the highest probability. CODE-NN implements the latter approach for code retrieval. NCS, by contrast, carries out search based only on words present in a code snippet manifestly, or added as described in Section 5.1. In future work we plan to evaluate the extent to which learned summaries, as created using the techniques mentioned in these papers, further increase the search quality.

8 Conclusion

Given the availability of large amounts of code, both in open source, as well as inside companies, an exciting possibility is to automatically answer Stack Overflow–like questions directly from code. We presented Neural Code Search (NCS), an end–to–end code search system that supports natural language queries.

The basic NCS relies on distance between continuous vector representations of code as well as queries to carry out search. We found that this model can be significantly enhanced by adding a layer of supervision to better map query words to words appearing in source code. We also found that a custom ranking approach can compensate for the *scattering* of results caused by a vector similarity based search.

We presented a case study of using NCS for answering questions from Stack Overflow on a code corpus downloaded from GitHub. NCS, which enhancements, is able to answer two-thirds of the questions in the curated Stack Overflow dataset correctly.

[3]The paper [2] refers to this as retrieval, but this is not retrieval of existing code.

Acknowledgements

We thank Maxim Sokolov and Edouard Grave for valuable discussions.

References

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference on Machine Learning (ICML)*.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 2123–2132.

[3] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 681–682.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* abs/1607.04606 (2016). arXiv:1607.04606 http://arxiv.org/abs/1607.04606

[5] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 10, 11 pages.

[6] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, Marsha Chechik and Martin Wirsing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–400.

[7] Zellig S. Harris. 1954. Distributional Structure. *WORD* 10, 2-3 (1954), 146–162. https://doi.org/10.1080/00437956.1954.11659520

[8] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *CoRR* abs/1702.08734 (2017). arXiv:1702.08734

[10] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In

Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, USA, 260–270.

[11] Chris J. Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, II–649–II–657. http://dl.acm.org/citation.cfm?id=3044805.3044965

[12] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 111–120.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546

[14] Bhaskar Mitra and Nick Craswell. 2017. Neural Models for Information Retrieval. *CoRR* abs/1705.01509 (2017). arXiv:1705.01509

[15] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR* abs/1703.05698 (2017). arXiv:1703.05698 http://arxiv.org/abs/1703.05698

[16] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 357–367.

[17] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 404–415.

[18] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017). arXiv:1709.00103 http://arxiv.org/abs/1709.00103