Exploring Input Factors in Neural Code Search

Anonymous Author(s)

ABSTRACT

abstract

KEYWORDS

keywords

ACM Reference Format:
Anonymous Author(s). 2019. Exploring Input Factors in Neural Code Search. In *Proceedings of The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

- *Research Question 1*: Do common input preprocessing steps improve performance?
- *Research Question 2*: Does vocabulary size impact code search performance?
- *Research Question 3*: What is the relationship between training data size and search performance?

2 CODE SEARCH AND NEURAL NETWORKS

Studies in the literature have found that code search is a major component in developers' tool-kits [20, 34, 35]. Effective code search can facilitate code re-use and maintenance, and improve the efficiency in software development [14, 28]. It is also found to be frequently performed, with on average 12 queries per workday for developers [30]. A major type of code search is where developers express their *intent* in natural language and use this to retrieve relevant code fragments, so as to find example code of how to do some tasks. A recent case study at Google shows that this type of code search is the most commonly used one (over a third of the cases) [30].

Historically, the effectiveness of this type of search relied on some amount of explicit overlap between the query and the retrieved results and have worked to weigh the overlap appropriately [26]. Code search techniques have also sought to incorporate additional, code-specific, information in the form of examples, test-cases, and type signatures [1, 12, 25]. Advances in neural networks have provided an additional alternative through learned representations that are designed to capture richer information by learning from a large training set. In particular, distributed representations in the form of real-valued vectors of complex objects has fueled advances in many areas, including natural language processing [6, 24, 27] and computer vision [7, 41]. There have been recent works in software engineering to apply distributed representations and neural models to code search and thus enable a richer approach to search that does not explicitly require token overlap. These works have demonstrated the potential of the neural approach to outperform classical techniques [13, 29]. These advances have led to the rise of techniques broadly labeled *neural code search*.

We focus on neural code search techniques that learn their representations from a training corpus of aligned code and natural language (NL) inputs. For example, a corpus of code results and their accompanying query would be a suitable corpus. However, such corpus is generally hard to obtain. In practice, developers have relied on scraping code fragments from large online repositories, such as GitHub, and pairing each fragment with its documentation or source comments [13, 29].

The architectures used to learn the appropriate representations vary, but the general framework is shared. To start with, we denote the raw code snippet as C and raw natural language as L. First, some form of preprocessing is applied on the raw code and natural language to obtain structured inputs $C_p = \phi_C(C), \;\; L_p = \phi_L(L)$, where $\phi_C(\cdot)$ and $\phi_L(\cdot)$ are the preprocessing operations. Then, some encoder models are used to compute the distributed representations (also referred to as *embeddings*) of the code and natural language, $\mathbf{v}_C = f_C(C_p)$, $\mathbf{v}_L = f_L(L_p)$. The encoders $f_C(\cdot)$ and $f_L(\cdot)$ are typically realized by some neural network models. We can then measure the relatedness between a code snippet and a natural language snippet by some similarity function in the embedding space: $s = \alpha(\mathbf{v}_C, \mathbf{v}_L)$. Common forms of the similarity function $\alpha(\cdot, \cdot)$ include cosine similarity, euclidean distance, or even another learnable model. Code search can be performed by encoding a user's search query into its embedding (using $f_L(\cdot)$), and searching an index, which contains available code fragments stored based on their embeddings (using $f_C(\cdot)$), and retrieving the entries with highest similarity (by $\alpha(\cdot, \cdot)$). Figure 1 shows a basic diagram summarizing this framework.

To train these models on aligned code-NL corpus, a wildly adopted method is using hinged loss with negative sampling [5, 8]. Concretely, for each NL snippet L, we construct a triplet $(L, C+, C-)$ where $C+$ is the code snippet aligned with L and $C-$ is a randomly sampled negative example from the pool of C in the corpus. Training is done by minimizing the loss:

$$\mathcal{L}(\theta) = \sum_{(L,C+,C-)\in P} \max(0, \delta - s_{L,C+} + s_{L,C-}) \qquad (1)$$

where θ denotes the parameters in the model, P is the training set, δ is a constant margin, and s is the similarity score from the model. Intuitively, this loss drives the similarity between the positive pair $(L, C+)$ to be larger than the negative pairs up to the margin δ. This loss is an approximation by sampling of the true ranking loss: in the limit of infinite samples, a zero loss on $\mathcal{L}$ will give perfect ranking on the training corpus (for each L as query, the aligned code $C+$ will be ranked No.1 among all the code snippets). It is worth pointing

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

out that previous work on code search has used negative sampling on queries instead of code, constructing triplet $(C, L+, L-)$ [13]. However, the loss constructed that way is not a valid approximation of the true ranking loss (the easiest way to see this is that a zero loss will not guarantee a perfect ranking on the training corpus, and vice versa). Therefore, we suggest the community to use the formulation presented here since it is theoretically grounded.

make black box clearer

3 PROCESSING INPUT FOR CODE SEARCH

Existing code search literature, and applications of machine learning to software engineering in general, often performs some amount of preprocessing on both the code and natural language inputs prior to training the model(s) used to learn representations. The goal of this preprocessing is generally to remove extraneous information and normalize the input. We briefly present some strategies from recent literature:

- *Deep Code Search* [13]: the authors derive three forms of input from the original code fragments: they split method names on camel case, they extract API invocation sequences, and they split other tokens, remove duplicates, stop words and key words. Their natural language input drops any parameter or return type annotations.
- *Learning to mine aligned code and natural language pairs from stack overflow* [39]: derives task-specific high level features for their code inputs and then splits natural language on white-space
- *Retrieval on source code: a neural code search* [29]: the authors extract qualified method names, method invocations, enumerations, string literals, but ignore variable names. All tokens are split on camel and snake case, lower cased, and certain tokens are removed based on their characters and length.
- *Code How: Effective Code Search Base on API Understanding and Extended Boolean Model* [23]: the authors extract the APIs in code fragments, the method body, and fully qualified method name. For natural language inputs, they remove punctuation, remove stop words, and stem tokens.

Despite the variety of possible preprocessing options, none of the research so far has presented a rigorous comparison of these options. In particular, it is not clear whether performance improves when the input is transformed, nor is it clear whether these transformations are suitable for general classes of models.

Other portions of the preprocessing pipeline have been equally ignored by current research. In particular, there has been little discussion of the possible impact of vocabulary size on the learned representations. Vocabulary size determines what tokens in the input are identified versus treated as a generic *Unknown* token. In practice, the vocabulary size can impact the generalizablity of the learned representation, where a large vocabulary may learn overly specific representations, while a small vocabulary may like the expressiveness required to appropriately model inputs.

Finally, neural code search papers in particular have emphasized the importance of a large training corpus, but have not performed a clear ablation study on the performance impact of differing amounts of training data. Collecting high quality training data can be a time and resource intensive task, so understanding the possible impact of less training data can help address the viability of applying neural code search to newer or more exotic code domains.

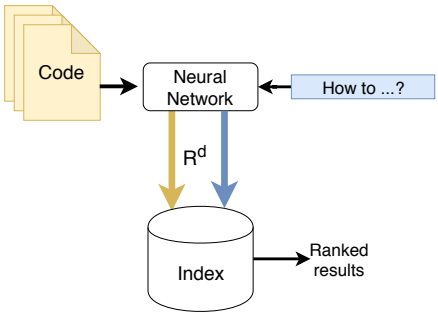

Figure 1: Neural code search generally works by computing a real-valued vector representation of code fragments available and natural language queries. These real-valued representations, known as *embeddings*, are designed to be similar for entries that share semantic overlap.

Figure 2 shows a stylized depiction of the training pipeline for neural code search models. Developers start by collecting a training corpus, typically by scraping existing open source repositories. They then prepare this corpus for the modeling step. Our study focuses on the steps between the start and end of this pipeline, namely the data preparation phase. To do so, we consider the following transformations as separate preprocessing operations and compose them to obtain different input processing pipelines.

- extracting method name and API calls from code inputs
- removing stop words and stemming tokens
- tokenizing inputs based on code-relevant punctuation
- removing parameter/type annotations from code documentation strings

In addition to these transformations, we experiment with a range of vocabulary sizes and training data sizes. We detail the exact configurations studied in Section 5.

4 NEURAL CODE SEARCH MODELS

In this section, we present a brief overview on the common choices of models for the encoders ($f_C(\cdot)$ and $f_L(\cdot)$) in code search. The encoder model takes a sequence of tokens as input and generates a vector as output. The model is typically in two stages: the first stage maps each input token to a token embedding by looking up in an embedding matrix, and the second stage is some compositional model that transforms a sequence of vectors into a single vector.

4.1 Token embedding

Each token in the vocabulary has a vector embedding associated with it. These embeddings are stored in a matrix, so the mapping from input tokens to token embeddings is done via matrix lookup. A common practice is to initialize the token embeddings with some pre-trained embeddings [17, 37]. When training the model on the task of interest, the token embeddings can either be kept fixed [31] or fine-tuned [37].

There are several methods for pre-training token embeddings... describe some examples including fasttext...

4.2 Compositional model

A vast range of compositional models have been proposed in literature, especially in natural language processing where they are used to build sentence or document embeddings from word embeddings [17, 21, 22, 37]. These models can be roughly categorized into two types: bag-based models and sequential models. The two types differ in whether the input tokens are treated as a ordered sequence or a set.

Bag-based models. This type of models treat the input tokens as a set, not taking into account the sequential order. The simplest among them is vector averaging, i.e. take the average of the token embeddings as the sequence embedding. Built upon that, models with additional networks over the averaged vector have been proposed.

Deep averaging network (DAN) is one of the popular models of this type [17]. Given a sequence of input token embeddings $\{\mathbf{x}_t\}_{t=1}^{T}$, DAN first compute the average $\mathbf{z}_0 = \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}_t$. Then this averaged vector is fed into a deep feed-forward network to obtain the output vector. Concretely, each layer computes

$$\mathbf{z}_i = g(\mathbf{z}_{i-1}) = f(\mathbf{W}_i\mathbf{z}_{i-1} + \mathbf{b}_i) \tag{2}$$

where $f(\cdot)$ is the activation function, $\mathbf{W}_i$ and $\mathbf{b}_i$ are the parameters of the i-th layer. The output from the last layer $\mathbf{z}_m$ is used as the sequence embedding.

There are other common bag-based models. [29] uses weighted average of input token embeddings with TF-IDF weights. [22] introduces self-attention as the mechanism for computing weights in the weighted average.

Sequential models. The second type of compositional models treats the input tokens as an ordered sequence. Recurrent neural network (RNN) based models is the major branch in this type. The general form of RNN is given by

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{3}$$

where $\mathbf{h}_t$ is the hidden state vector and $\mathbf{x}_t$ is the input vector, both at the t-th step in the sequence. The hidden state is recurrently computed using the previous-step hidden state and the current input. Different forms of computation $g(\cdot)$ gives rise to different variants of RNN, among them the most popular ones are long short-term memory (LSTM) [15] and gated recurrent unit (GRU) [4]. We will briefly describe here bi-directional LSTM model.

Bi-directional LSTM (Bi-LSTM). One form of the recurrent computation of LSTM is given by [10]:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(W_{ii}\mathbf{x}_t + \mathbf{b}_{ii} + W_{hi}\mathbf{h}_{(t-1)} + \mathbf{b}_{hi}) \\
\mathbf{f}_t &= \sigma(W_{if}\mathbf{x}_t + \mathbf{b}_{if} + W_{hf}\mathbf{h}_{(t-1)} + \mathbf{b}_{hf}) \\
\mathbf{g}_t &= \tanh(W_{ig}\mathbf{x}_t + \mathbf{b}_{ig} + W_{hc}\mathbf{h}_{(t-1)} + \mathbf{b}_{hg}) \\
\mathbf{o}_t &= \sigma(W_{io}\mathbf{x}_t + \mathbf{b}_{io} + W_{ho}\mathbf{h}_{(t-1)} + \mathbf{b}_{ho}) \\
\mathbf{c}_t &= \mathbf{f}_t * \mathbf{c}_{(t-1)} + \mathbf{i}_t * \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t * \tanh(\mathbf{c}_t)
\end{aligned}$$

This should be short but enough for people with little background to understand our experiments

- Sequence vs bag
- give self-contained tutorial on sequence based models (bilstm)
- give self-contained tutorial on bag based models (dan)
- give self-contained tutorial on unsupervised word embedding computation

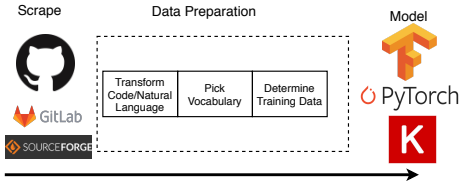

Figure 2: Our study focuses on the portion of the neural code search pipeline in the bounding box. This includes the impact of different code/natural language transformations, the vocabulary size, and amount of training data on the eventual performance of a neural code search system.

5 EXPERIMENTAL METHODOLOGY

5.1 Training and Evaluation Data

For training purposes, we use a corpus of approximately 1 million Python function definitions and corresponding documentation strings released by GitHub and used to train a sequence-based neural code search system [11]. We removed any function that did not successfully parse with the standard Python parser.

For evaluation, we rely on two separate benchmark sets. The first test dataset consists of a randomly sampled set of 500 function definitions and docstring pairs from the GitHub test data (a separate dataset from the training data used). Two authors separately validated these entries to verify that the accompanying docstring successfully describes the function, removing any function/docstring pair where the correspondence was not clear. For our second test dataset, we use the test data released as part of *The Code/Natural Language Challenge* [40]. This test data consists of 500 Python code snippets with an accompanying *intent* description in natural language. This evaluation dataset is characterized by short inputs and was manually annotated to facilitate evaluation of systems that "generate program snippets from natural language". We refer to these two benchmark sets as *CONALA* and *GitHub*, respectively, throughout the remainder of the paper.

The *GitHub* test data is meant to measure in-domain performance, where the test data can be expected to be reasonably well represented by the available training data. In this case, that means that both training and test observations contain full function definitions and accompanying documentation strings. In contrast, the *CONALA* test data is meant to measure out-of-domain performance, given that the model was trained on the Github training corpus. In this case, the test data consists of code snippets rather than full function definitions, and curated snippet intents rather than documentation strings.

5.2 Models

david describe the bi-lstm, dan con-

5.3 Experiments

5.4 Evaluation Metrics

6 RESULTS

6.1 Performance of Classical Techniques

* put result from BM25 (and maybe TF-IDF if possible, timewise) to show that the task is not easy

6.2 RQ1: Preprocessing

Our experiments show that a simple preprocessing strategy for the code and natural language inputs yields the best performance across both *biLSTM* and *DAN* models under the out-of-domain (*CONALA*) and in-domain (*GitHub*) evaluations. In particular, removing information from the original input, by considering only a subset of the tokens available, actually led to lower performance in all cases.

6.2.1 Code transforms. We present the results for a subset of the pipelines we evaluated. In particular, we focus on:

- *Tokenize*: splits into tokens on punctuation, math operators, camel and snake case and lower-cases all tokens
- *Only method names + calls*: Tokenizes only portions of the input, the parts associated with the name of the function definition and any calls in the code snippet.
- *Tokenize + remove stop words*: Applies *Tokenize* and then removes English stop-words
- *Tokenize + remove stop-words + stem*: Applies *Tokenize + remove stop words* and then stems tokens based on the Porter stemmer

In all cases, the natural language treatment was kept constant, removing parameter and return annotations, tokenizing inputs and lower casing all tokens. For test data, we applied the appropriate pipeline, but if no tokens were produced (e.g. a *CONALA* code snippet with no function calls when applying *Only method names + calls*), we fall back to simply tokenizing the original input for that entry.

Table 1 and Table 2 present our results. In both cases, we see that considering only method names and calls lowers performance significantly. The best pipeline for both models was to perform tokenization and then remove English stop words.

6.2.2 Natural Language transforms.

- Table
- Describe findings, each option useful/not matter/bad, in different cases (code long vs short, model sequential vs not). Interpretation and understanding.

6.3 RQ2: Vocabulary size

A key decision for neural code search developers is the size of the vocabulary that they choose to represent. All tokens found during training or testing that are not in the vocabulary are typically represented by a catch-all value.

Table 1: Only considering method name and calls in the function bodies led to lower performing *biLSTM* in both evaluation datasets. The effect was larger when using the in-domain test data (*GitHub*). The most effective pipeline was *Tokenize + remove stop words* which splits words on characters, camel and snake case, and then removes English stop-words.

Pipeline	MRR	Acc@1	Acc@5	Acc@10
Tokenize	0.24 (0.01)	0.14 (0.01)	0.33 (0.01)	0.46 (0.01)
Only method name + calls	0.21 (0.01)	0.11 (0.01)	0.31 (0.01)	0.43 (0.02)
Tokenize + remove stop words	**0.25 (0.00)**	**0.15 (0.01)**	**0.36 (0.01)**	**0.48 (0.01)**
Tokenize + remove stop words + stem	0.24 (0.00)	0.14 (0.00)	0.33 (0.01)	0.46 (0.01)

(a) *CONALA*

Pipeline	MRR	Acc@1	Acc@5	Acc@10
Tokenize	0.66 (0.01)	0.56 (0.02)	0.79 (0.01)	0.85 (0.01)
Only method name + calls	0.43 (0.02)	0.32 (0.02)	0.55 (0.03)	0.64 (0.02)
Tokenize + remove stop words	**0.68 (0.01)**	**0.58 (0.01)**	**0.80 (0.01)**	**0.86 (0.01)**
Tokenize + remove stop words + stem	0.63 (0.01)	0.52 (0.01)	0.75 (0.01)	0.82 (0.01)

(b) *GitHub*

Table 2: Similar to the effect observed when using an *biLSTM*, the *DAN* model performance degraded when we only considered method name and calls. The drop in performance was larger when evaluating on the in-domain dataset (*GitHub*).

Pipeline	MRR	Acc@1	Acc@5	Acc@10
Tokenize	0.18 (0.01)	0.10 (0.01)	0.25 (0.01)	0.36 (0.01)
Only method name + calls	0.16 (0.01)	0.07 (0.01)	0.23 (0.02)	0.33 (0.03)
Tokenize + remove stop words	**0.19 (0.01)**	**0.11 (0.01)**	**0.27 (0.01)**	**0.39 (0.01)**
Tokenize + remove stop words + stem	0.18 (0.01)	0.10 (0.01)	0.26 (0.02)	0.37 (0.02)

(a) *CONALA*

Pipeline	MRR	Acc@1	Acc@5	Acc@10
Tokenize	0.50 (0.01)	0.38 (0.01)	0.63 (0.01)	0.72 (0.01)
Only method name + calls	0.31 (0.02)	0.21 (0.02)	0.42 (0.03)	0.53 (0.03)
Tokenize + remove stop words	**0.50 (0.02)**	**0.38 (0.02)**	**0.63 (0.03)**	**0.73 (0.02)**
Tokenize + remove stop words + stem	0.46 (0.01)	0.33 (0.02)	0.60 (0.01)	0.70 (0.01)

(b) *GitHub*

Our results show that performance is robust to large reductions in vocabulary size. Given that many tokens are rare, a smaller vocabulary, while able to represent a signficantly smaller part of the domain, in practice can cover a substantial fraction of the inputs' tokens.

Figure 3 shows the possible reductions in vocabulary size based on different token frequency cutoffs. A large fraction of the vocabulary occurs less than 10 times in the training data.

Table 3 presents the Acc@10 results for *biLSTM* on both *CONALA* and *GitHub*. Performance is robust despite vocabular reductions. Acc@10 only starts to degrade once the cutoff is substantially higher. Other performance metrics, and performance in *DAN*, display this same behavior.

Table 4 shows that reducing the vocabulary size, which we saw did not have a substantial impact on performance, can produce significant reductions in the memory required to represent the embeddings.

mention the initializing of embeddings where

describe experiment configurations in detail

describe sampling used to run pipelines multiple times, given time to train

summary table describing configurations for experiments

describe mrr, answered @1/5/10, and why this type of eval-

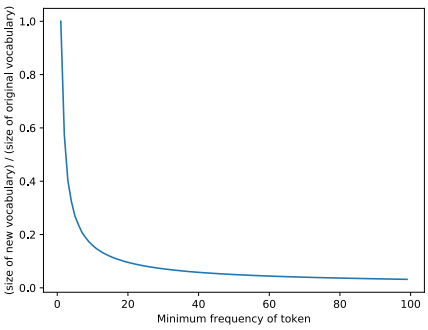

Figure 3: A large fraction of the tokens in the training data are rare tokens, appearing less than 10 times. A standard preprocessing pipeline must determine the target size of the vocabulary for which to learn embeddings.

Table 3: *biLSTM* Acc@10 for both test data sets. Performance remains robust for increases in minimum token frequency, which reduce vocabulary, until this cutoff is set substantially higher. Same holds true for other performance metrics in both models.

Minimum Frequency	*CONALA* Acc@10	*GitHub* Acc@10
1	0.45 (0.02)	0.84 (0.01)
10	0.46 (0.01)	0.86 (0.01)
20	0.47 (0.01)	0.84 (0.01)
100	0.46 (0.02)	0.82 (0.01)
1000	0.42 (0.01)	0.77 (0.01)

Table 4: Increasing the minimum token frequency can reduce the vocabulary size substantially, and result in a much smaller in-memory tensor required to represent the per-token embeddings. Memory measured for tensors in Pytorch (version 0.4.1.post2).

Minimum Frequency	Vocabulary Size	In-memory Tensor Size
1	635k	254MB
10	101k	40MB
20	61k	24MB
100	20k	8MB
1000	4.9k	2MB

6.4 RQ3: Training data size

Recent neural code search papers emphasize training on large datasets , on the order of millions of aligned code and natural language pairs. However, prior work has not presented a clear study of the impact of training data size on search performance.

Our results show that XXX .

We re-trained the best performing *DAN* and *biLSTM* models using the complete training dataset (approximately 1 million examples), rather than repeated downsampled subsets. We reduced the number of training epochs to 10, as each epoch now iterates over significantly more examples. As in prior experiments, we continue to choose the best performing model based on a held-out validation dataset. **??** shows that all evaluation metrics in both models increased when training on the larger dataset.

Table 5: Training with the full dataset, rather than downsampled subset, increased performance across all metrics and models. The sequence-based *biLSTM* displayed larger improvements.

Evaluation	Training Dataset	MRR	Acc@1	Acc@5	Acc@10
CONALA	Downsampled	0.25	0.15	0.36	0.48
CONALA	Full	0.3	0.18	0.41	0.56
GitHub	Downsampled	0.68	0.58	0.8	0.86
GitHub	Full	0.82	0.75	0.9	0.93

(a) *biLSTM* results

Evaluation	Training Dataset	MRR	Acc@1	Acc@5	Acc@10
CONALA	Downsampled	0.19	0.11	0.27	0.39
CONALA	Full	0.23	0.13	0.33	0.45
GitHub	Downsampled	0.5	0.38	0.63	0.73
GitHub	Full	0.56	0.45	0.68	0.78

(b) *DAN* results

To explore the relationship between training set size and performance we carried out additional experiments, where we downsampled the training data to increasingly larger subsets. **??** shows that XXX.

- full dataset gives better performance across the board
- large dataset is expensive to collect though
- so let's start from 10k up to full and characterize performance with curves
- can we come up with a general rule of thumb?

7 THREATS TO VALIDITY

In practice, code search is typically carried out over large corpora, some times in the millions , and results are subjectively rewarded based on their relevance to the query. Our experiments relied on two test sets of 500 code/natural language pairs. For each entry in our test data, we treated the natural language input as a user query and the corresponding code pair as the ground truth result. We then computed metrics such as MRR and accuracy at different cutoffs. While this evaluation differs from a production scenario, it allowed us to experiment with significantly more configurations and produce reproducible evaluation results. We believe that this controlled setting was key to deriving generalizable insights.

Our study focused on two model types: a sequence-based biLSTM and a bag-based DAN network. While other architectures may be used in code search, we believe that these two networks are representative of some of common techniques in use in state-of-the-art neural code search research, and are likely to remain relevant components in future systems given their ease of development and use.

cite

write what we observe here once we complete the experiments

We relied on a training corpus collected by scraping public Python repositories. To the extent that these repositories are particularly idiosyncratic, the observations made in our experiments may not generalize. However, given that the corpus was collected independently of our study and contains a large number of repositories and source files, we believe that this is not likely issue.

8 RELATED WORK

Code search has long been an active area of software engineering. Previous systems have explored the use of semantic information for querying. Reiss [28] presented a semantic code search engine that unified search techniques through the concept of specifications for the search target. A specification could be the type signature of the expected code snippet, keywords, input/output examples, test cases, amongst others. DemoMatch [38] presented the idea of retrieving relevant code snippets by demonstrating the desired functionality in a different application. This type of trace-based querying relies on a database of previously executed traces, rather than code snippets. Other research directions have included using code search to enhance other productivity tools, such as documentation. For example, eXoaDocs [18] creates a code search index where entries are augmented with automatically collected example usage, users could then search for a specific API and obtain high quality examples in return.

Previous classical code search systems have also explored the use of natural language queries in retrieval. SNIFF [3] annotated source code with API documentation to allow natural language querying of code snippets and leveraged a type-based intersection of candidate results to produce a smaller set for the user. CodeHow [23] similarly extends a users query with potential candidate APIs and then uses the extended search query to retrieve candidates using an extended boolean model.

Applications of neural networks to software engineering has become increasingly popular. Neural code search systems, such as NCS [29] and CODEnn [13], use different neural architectures to learn embeddings for code and user queries. Their effectiveness, and distinctive input processing steps, motivated this research paper. Other related applications of neural networks include program synthesis from free-form queries [36], improving program synthesis from input-output examples [33], and code summarization [16].

Data preprocessing for learning has been recognized as an important factor in performance across multiple domains [2, 9, 19, 32]. However, to the best of our knowledge this is the first study investigating the effect of different preprocessing decisions on neural code search techniques.

9 CONCLUSION

10 TOEDIT LIST (REMOVE WHEN FINALIZED)

- Where to mention negative sampling argument, and shall we include quantitative results
- Better naming of title and sections
- Check more carefully on paper format and conform to it.

REFERENCES

[1] 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*. ACM, 681–682.

[2] Barbara Cannas, Alessandra Fanni, Linda See, and Giuliana Sias. 2006. Data preprocessing for river flow forecasting using neural networks: wavelet transforms and data partitioning. *Physics and Chemistry of the Earth, Parts A/B/C* 31, 18 (2006), 1164–1171.

[3] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 385–400.

[4] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[5] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2493–2537. http://dl.acm.org/citation.cfm?id=1953048.2078186

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Tomas Mikolov, et al. 2013. Devise: A deep visual-semantic embedding model. In *Advances in neural information processing systems*. 2121–2129.

[8] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Marc' Aurelio Ranzato, and Tomas Mikolov. 2013. DeViSE: A Deep Visual-Semantic Embedding Model. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2121–2129. http://papers.nips.cc/paper/5204-devise-a-deep-visual-semantic-embedding-model.pdf

[9] Salvador García, Julián Luengo, and Francisco Herrera. 2015. *Data preprocessing in data mining*. Springer.

[10] F. A. Gers, J. Schmidhuber, and F. Cummins. 1999. Learning to forget: continual prediction with LSTM. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, Vol. 2. 850–855 vol.2. https://doi.org/10.1049/cp:19991218

[11] Github. [n. d.]. Semantic Code Search. https://github.com/hamelsmu/code_search/blob/master/notebooks/1-PreprocessData.ipynb

[12] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. 2010. Exemplar: EXEcutable exaMPLes ARchive. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 259–262. https://doi.org/10.1145/1810295.1810347

[13] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 933–944.

[14] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2009. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 232–242. https://doi.org/10.1109/ICSE.2009.5070524

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[16] 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)*, Vol. 1. 2073–2083.

[17] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 1681–1691. https://doi.org/10.3115/v1/P15-1162

[18] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an Intelligent Code Search Engine.. In *AAAI*.

[19] R Lacroix, F Salehi, XZ Yang, and KM Wade. 1997. Effects of data preprocessing on the performance of artificial neural networks for dairy yield prediction and cow culling classification. *Transactions of the ASAE* 40, 3 (1997), 839–846.

[20] Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in*

Computing Systems. ACM, 1323–1332.

[21] Jiwei Li, Thang Luong, and Dan Jurafsky. 2015. A Hierarchical Neural Autoencoder for Paragraphs and Documents. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1106–1115. http://www.aclweb.org/anthology/P15-1107

[22] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A Structured Self-Attentive Sentence Embedding. https://openreview.net/forum?id=BJC_jUqxe

[23] 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. https://doi.org/10.1109/ASE.2015.42

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[25] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Acm Sigplan Notices*, Vol. 47. ACM, 997–1016.

[26] Jiaul H. Paik. 2013. A Novel TF-IDF Weighting Scheme for Effective Ranking. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '13)*. ACM, New York, NY, USA, 343–352. https://doi.org/10.1145/2484028.2484070

[27] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365* (2018).

[28] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 243–253. https://doi.org/10.1109/ICSE.2009.5070525

[29] 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 the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, New York, NY, USA, 31–41. https://doi.org/10.1145/3211346.3211353

[30] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[31] Darsh Shah, Tao Lei, Alessandro Moschitti, Salvatore Romeo, and Preslav Nakov. 2018. Adversarial Domain Adaptation for Duplicate Question Detection. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1056–1063. http://aclweb.org/anthology/D18-1131

[32] Marnie E Shaw, Stephen C Strother, Maria Gavrilescu, Katherine Podzebenko, Anthony Waites, John Watson, Jon Anderson, Graeme Jackson, and Gary Egan. 2003. Evaluating subject specific preprocessing choices in multisubject fMRI data sets using data-driven performance metrics. *NeuroImage* 19, 3 (2003), 988–1001.

[33] Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *Advances in Neural Information Processing Systems*. 8931–8940.

[34] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 1 (2011), 4.

[35] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 26.

[36] Chenglong Wang, Po-Sen Huang, Alex Polozov, Marc Brockschmidt, and Rishabh Singh. 2018. Execution-guided neural program decoding. *arXiv preprint arXiv:1807.03100* (2018).

[37] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2015. Towards Universal Paraphrastic Sentence Embeddings. *CoRR* abs/1511.08198 (2015). http://arxiv.org/abs/1511.08198

[38] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API discovery from demonstrations. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 64–78.

[39] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[40] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM, 476–486. https://doi.org/10.1145/3196398.3196408

[41] Quanzeng You, Zhengyou Zhang, and Jiebo Luo. 2018. End-to-End Convolutional Semantic Embeddings. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.