Exploring Input Factors in Neural Code Search

ABSTRACT

abstract

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

keywords

ACM Reference Format:

. 2018. Exploring Input Factors in Neural Code Search. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

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

A major goal of code search has been to enable developers to express their *intent* in natural language and use this to retrieve relevant code fragments. Effective code search can facilitate code re-use and maintenance [7, 14]. Studies in the literature have found that code search is a major component in developers' tool-kits [8, 17, 18]. A recent case study at Google finds continued support for this trend, observing that that developers searched code frequently, up to 12 queries per workday [16].

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 [12]. Code search techniques have also sought to incorporate additional, code-specific, information in the form of examples, test-cases, and type signatures [1, 5, 11]. 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 natural language processing [2, 10, 13] and computer vision [3, 21].

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

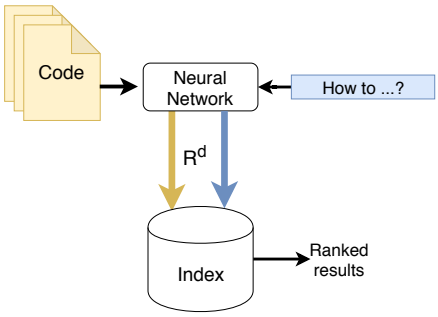

Figure 1: Neural code search generally works by computing a real-valued vector representation of code fragments available and natural language queries. These real-valued representations, known as *embeddings*, are designed to be similar for entries that share semantic overlap.

Recent work in software engineering has worked to apply embeddings to code search and thus enable a richer approach to search that does not explicitly require token overlap. Papers, such as [6, 15], have demonstrated the ability to outperform classical techniques. 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 inputs. For example, a corpus of code results and their accompanying query would be a suitable corpus. 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 [6, 15]. The architectures used to learn the appropriate representations vary, but the general framework is shared: the network learns a function that maps code input to a real-valued vector, natural language input to a real-valued vector, and some similarity metric over the two vectors is maximized for entries that are aligned. Code search can then be performed by *encoding* a user's search query into its corresponding real-valued vector, and searching an index, which contains available code fragments stored based on their real-valuated representations, and retrieving the entries with highest similarity. Figure 1 shows a basic diagram summarizing this framework.

- Mention the correct negative sampling method.

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 (e.g. programming language keywords) and normalize the input. We briefly present some strategies from recent literature:

- *Deep Code Search* [6]: 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* [19]: 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* [15]: 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* [9]: 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 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

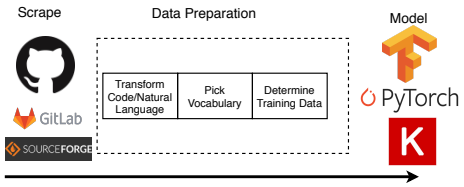

Figure 2: Our study focuses on the portion of the neural code search pipeline in the bounding box. This includes the impact of different code/natural language transformations, the vocabulary size, and amount of training data on the eventual performance of a neural code search system.

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

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 basesd models (dan)
- give self-contained tutorial on unsupervised word embedding computation

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 [4]. 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* [20]. 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 configurations

5.3 Experiments

5.4 Evaluation Metrics

6 RESULTS

6.1 Performance of Classical Techniques

describe why we chose these two types of models

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

6.2 RQ1: Preprocessing

- 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

- Table, graph
- Discuss findings.

6.4 RQ3: Training data size

mention the initializing of embeddings where

- 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

describe experiment configurations in detail

- you evaluate using 500 conala/github questions, real-world code search is over millions of entries
  - Answer: allows us to compare configurations in controllable environment and without manual annotation, comparing to ground truth so also allows reproducibility
- you evaluate only two types of models
  - yes, but they are representative of two main model classes in current code search
- different corpora may have different results
  - evaluated on two datasets for this exact reason, believe we are touching two ends of code search (short snippet vs full method)

8 RELATED WORK

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

- related code search
- related neural code search
- related pre-processing impact on other neural applications

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] 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).

[3] 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.

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

[5] 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

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

[7] 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

[8] 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.

[9] 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

[10] 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.

[11] 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.

[12] 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

[13] 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).

[14] 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

[15] 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

[16] 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.

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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)*.