Semantic Code Search with Neural Bag-of-Words and Graph Convolutional Networks

Anna Abad Sieper[1] Omar Amarkhel[2] Savina Diez[3] Dominic Petrak[4]

Abstract: Software developers are often confronted with tasks for which there are widespread solution patterns. Searching for solutions using natural language queries often leads to unsatisfying results. Github, Microsoft Research and Weights & Biases created the CodeSearchNet Challenge [Hu19] to address this problem. Its goal is to develop code search approaches that return the code that best matches a natural language query. In this paper, we investigate two different approaches in this context. First, a Neural Bag-of-Words encoder using TF-IDF weighting and second, a Graph Convolutional Network which includes the call hierarchy in a target method's representation. In our experiments we were able to improve the Neural Bag-of-Words models, whose results were published in the CodeSearchNet Challenge. Our Neural Bag-of-Words encoder improves the MRR by 4.38% for Python and 4.98% for Java. The Graph Convolutional Network did not improve the results over of the Neural Bag-of-Words model.

Keywords: Semantic Code Search; Graph Convolutional Network; Neural Bag-of-Words; CodeSearchNet Challenge

1 Introduction

This paper introduces two novel approaches towards the development of a search engine for code. This challenge - referred to as semantic code search - has applications in supporting software development because it faciliates an easier on-demand reuse of code. Github, Microsoft Research and Weights & Biases have established the CodeSearchNet Challenge [Hu19] for this purpose. The participants are asked to design their own approaches to retrieve a suitable code function from a corpus. The provided data corpus includes code functions from open source Github repositories and their docstrings. It contains code in the programming languages Python, Java, Ruby, Go and PHP. [5]

Semantic similarities between search queries and code functions are not always based exclusively on the same keywords, for example when looking at synonyms such as "picture" and "image" and declensions such as "load" and "loading". Instead of using simple keyword matching, semantic relationships between words should be learned.

One common approach called *Bag-of-Words* is to be developed. Bag-of-Words models map

[1] Hochschule RheinMain, Wiesbaden, Germany a.abadsieper@gmail.com
[2] Hochschule RheinMain, Wiesbaden, Germany omar.m.amarkhel@gmail.com
[3] Hochschule RheinMain, Wiesbaden, Germany savinadz@gmail.com
[4] Hochschule RheinMain, Wiesbaden, Germany dominicpetrak@googlemail.com
[5] https://github.blog/2019-09-26-introducing-the-codesearchnet-challenge/

search queries and code functions each to a vector that marks the keywords. By means of neural networks, similarities between vectors belonging to each other (code function and corresponding docstring) are to be established. Ultimately, syntactically different words that are, however, in context with each other are to be identified. For example, it could be learnt that queries containing the word “database” match functions containing the word “db”.
Our second approach deals with the inclusion of code that is called within code functions. For this purpose, caller graphs are used. They are assembled based on the corresponding Github repositories and consumed by a Graph Convolutional Network. With the help of this technique, the features of a function are to be extended by the features of its callees. Ultimately, functions are to be characterized more extensively in this way.
We make the following contributions:

- A Graph Convolutional Network trained on caller graphs is used for semantic code search.
- A Neural Bag-of-Words method with weighting schemes for identifying semantic similarity between search query and code function is used to retrieve code.
- The models are evaluated quantitatively and compared to baseline models from the CodeSearchNet Challenge. Our findings are that TF-IDF weighting and Byte Pair Encoding bring substantial improvement to the Neural Bag-of-Words approach.

Starting with related approaches, the CodeSearchNet Challenge will be presented and explained in more detail. Then the data corpus used is presented. In the following chapter we present our approaches to graph-based data processing and to identifying similarities between docstring and code using the Bag-of-Words method. Finally, the results of our experiments with the presented methods are shown and a brief conclusion is given.

2 Foundations

This work addresses the CodeSearchNet Challenge and the neural learning of graph representations. A short overview of related work is given below.

2.1 CodeSearchNet Challenge

Semantic code search is the task of finding relevant code given a natural language query. It requires bridging the gap between natural language and the language used in the program code. Unlike natural language, code is not created to be read. Code contains purely syntactic tokens and different identifiers for the same content. The goal is to relate identifiers in code, such as “db”, to terms like “database” in natural language.

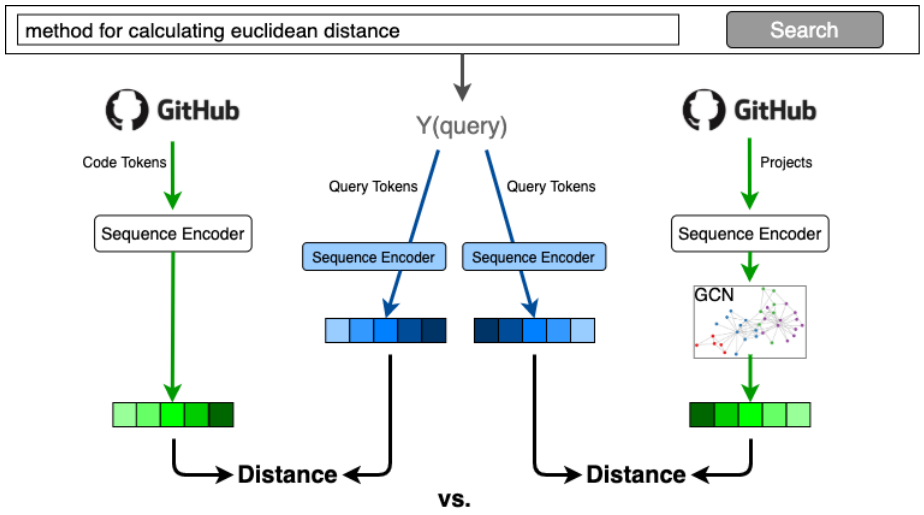

Fig. 1: We suggest two approaches towards code search (left/right). In both approaches, queries (blue) and the respective code (green) from Github are processed by an encoder. Then the respective pairs are matched based on the distance (bottom).

With today's standardized code documentation, both the code and natural language descriptions can be extracted for each documented method of a software project, resulting in data records containing pairs of code and language. This forms an interesting basis for learning code-language similarity measures at the large. To encourage researchers and practitioners to further explore this interesting task, the CodeSearchNet Challenge was created and presented by Github, Microsoft Research and Weights & Biases as a competition.

The CodeSearchNet Challenge provides an extensive corpus of training, test and validation data. Overall it consists of 2 million (comment, code) pairs crawled from different open source libraries, covering six widely used programming languages: Python, JavaScript, Java, Ruby, Go and PHP. The dataset is provided as json lines files, each containing json objects of (comment, code) pairs in a preprocessed manner [Hu19]. This enables participants to directly reuse them for building representations and training, testing and validation of machine learning approaches like neural networks.

2.2 Related Works on Retrieving Code

There exist various approaches on retrieving code. In this section, some related works are mentioned that dealt with a similar topic.
In "Semantics-Based Code Search" by Reiss et. al [Re09], relevant source code snippets are suggested based on the specification formulated by the user. This specification must at best include semantics and signature and should be formulated as precisely as possible.

The initial suggestions are selected by keyword matching. To have a more robust system with regard to synonyms, our approach learns semantic connections between terms. Wan et. al's approach in their paper "Multi-Modal Attention Network Learning for Semantic Source Code Retrieval" [Wa19] uses Abstract Syntax Trees and Control Flow Graphs for code embeddings. Furthermore, the system works with an LSTM and a Gated Graph Neural Network. An abstract syntax tree is used to decompose code fragments syntactically in order to better extract the semantics. In contrast, we only use the code tokens. The approach of Kanade et. al, discussed in "Pre-trained Contextual Embedding of Source Code" [Ka19], proposes CuBERT (Code Understanding BERT) as a further development of the BERT framework. Here Word2Vec embeddings for code are pre-trained. As in this paper, in Sachdev et. al's paper "Retrieval on Source Code: A Neural Code Search" [Sa18], word embeddings are used in combination with TF-IDF weighting. Furthermore, a high-dimensional vector similarity search is performed. Additionally, a supervision layer and a custom ranking system have been developed. In Cambronero et. al's work "When Deep Learning Met Code Search" [CKC], the authors used a Sequence-to-Sequence Gated Recurrent Unit Network and an LSTM for code retrieval.

2.3 Graph Convolutional Network (GCN)

The key to semantic code search is to understand the complete functionality of a method. However, methods are usually not functionally complete but may outsource vast parts of their functionality to other methods, so that the internal dependencies of methods become relevant. To represent the functional range of a method, it is common practice to represent software programs as caller graphs [NGV08]. A Graph Convolutional Network can be applied to the resulting graph-structured data. Our key idea is to utilize this caller graph to learn more complete representations of methods for semantic code search.

A GCN is a folding neural network for learning on graph structured data. The most prominent approach for this task is provided by Kipf et al. [KW16] and offers a modified architecture for spectral graph convolution on undirected graphs. Each layer can be described as follows:

$$H^{l+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{l}W^{l}) \tag{1}$$

In 1, $\hat{A} = A + I$ is the graph's adjacency matrix with added self-connections and I as identity matrix. $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$. W^l is a layer-specific trainable weight-matrix. Input parameters for H^{l+1} are H^l and A, with $H^0 = X$ and X as feature matrix. $\sigma(\cdot)$ is a non-linear activation function (in our case ReLU, Rectified Linear Unit). The GCN proposed by Kipf et al. shows a significant increase in efficiency and improvement in the classification of nodes and graphs, compared to earlier work in this field [KW16].

3 Approach

The general approach towards semantic code search is to encode a search query and code function tokens obtaining embeddings. Then the resulting embeddings are compared, obtaining a similarity. The following notation is defined:

- Given a search query q and a target function f, a sample is designated (q, f).
- The vocabulary that is created using the training samples and should contain all relevant words is referred to as V. It consists of the preprocessed natural language tokens and code tokens.
- A word (or token) is denoted by t. This can belong to a query or a method. A list of relevant tokens from a query or a method, also referred to as a document below, is denoted by $d = [t \mid t \in d \text{ and } t \in V]$.
- The number of occurrences of a word t in document d is represented as $\#(t, d)$.
- The embedding vector of a document d is a real-valued vector and denoted by e_d.

Given a query q, we rank functions f by the similarity between e_q and e_f.

3.1 Bag-of-Words

As a baseline a *Bag-of-Words* encoder is used. For this purpose, a vocabulary is generated from the most frequently occurring words in the entire corpus. Each document gets an embedding $e_d \in \mathbb{N}^V$, where $e_d[i] = \#(t_i, d)$ and $t_i = V[i]$. Finally, the distances of an encoded query to the encoded code functions are measured by the angular metric. The smaller the distance, the better the function is ranked. In the following, the implemented model is referred to as BoW (Bag-of-Words) encoder.
The data from the CodeSearchNet Challenge is preprocessed by splitting at special characters, camelcase splitting as well as lowercasing and stopword filtering. The stop word list contains the keywords of the respective programming language. For example, the document $d =$ [def, connect_to_db(portNumber)] is transformed to $d' =$ [connect, to, db, port, number].

In further experiments the preprocessing is extended by a Byte Pair Encoding (BPE). For this purpose, the vocabulary from the training data is created and stored using a Sentencepiece[6] model. Before creating the embeddings, the tokens are preprocessed using this model.

In order to distinguish relevant words from unimportant ones, words can also be weighted using the TF-IDF (Term Frequency-Inverse Document Frequency) scheme. The weight of

[6] https://github.com/google/sentencepiece

a word in the entire corpus tfidf(t,d) is thus calculated as shown below and normalized to euclidean norm [De20].

$$\text{tfidf}(t, d) = \#(t, d) \cdot log \frac{\#d + 1}{\sum \mathbf{1}_{t \in D} + 1} \tag{2}$$

The idea of this formula is to combine the frequency of a token t in the document d and the relative frequency of the documents in which t occurs into a total weight.
The vocabulary is created from all tokens in all documents. The embedding e_d contains the TF-IDF values in the order of the words in the vocabulary. Frequent tokens get low weights anyway, so a stopword list is omitted. The encoder operating with the TF-IDF weights is referred to as BoW TF-IDF encoder.

3.2 Neural Bag-of-Words

To improve the results of the BoW TF-IDF encoder, we aimed for a fuzzy matching (e.g. image matches picture), which we implemented with a Neural Bag-of-Words encoder (NBoW). Each token $t \in V$ gets an embedding $e_\text{t} \in \mathbb{R}^\text{k}$ of k dimensions. The embedding of a docstring or a method is calculated by averaging its tokens' embeddings. In addition, TF-IDF weights are used to weight the embeddings accordingly:

$$e_\text{d} = \frac{1}{\#d} \sum_{t \in d} \text{tfidf}(t, d) \cdot e_\text{t} \tag{3}$$

The embeddings e_t are trained such that matching code and queries are considered similar while unmatching code and queries are not. The Triplet Margin Loss [VM16] is used for this in training. It takes a docstring embedding as anchor e_a, its corresponding code embedding as positive example e_p and the code embedding of a random other sample as negative example e_n. A margin γ is also used, which is set to 1 by default.

$$L(a, p, n) = max\{||e_a - e_p||_2 - ||e_a - e_n||_2 + \gamma, 0\} \tag{4}$$

Basically, the loss is zero if e_a and e_p have a short distance, but e_a and e_n have a large distance.

3.3 GCN-based Neural Bag-of-Words

As a novel approach towards code representation we use Graph Convolutional Networks. To represent the functionality of a method as accurately as possible, it is important to understand all other methods called in the method under consideration. To view internal

dependencies, we consider a method as a node in a caller graph [NGV08]. For learning on this kind of data we use the modified framework of spectral graph convolutions by T. Kipf and M. Welling [KW16]. Since this approach can only work on undirected graphs, we also consider caller graphs as undirected graphs.

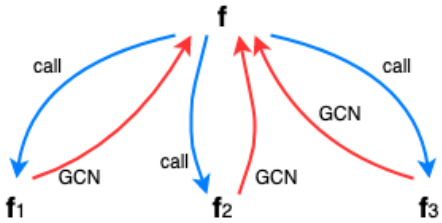

Fig. 2: The blue edges show which methods are called by f. In order to be able to display the functionality of f as accurately as possible, f must be enriched with the information of all called methods. The GCN (red edges) aggregates the features of the called methods up to f.

To train the GCN, the method's code is extracted from the caller graphs. To apply the propagation rule from Equation 1, the code is entered as H. The caller graph's adjacency matrix is used as A. As loss function, the Triplet Margin Loss is derived from Equation 4 is used, where it takes the output of the GCN as anchor, its corresponding docstring embedding as positive example and the docstring embedding of a random sample as negative example. In this way, the net tries to approximate the natural language description of a method and its code. We trained the GCN on 4.292 caller graphs (see Table 1), 100 iterations each, using a learning rate of 0.004 and Adam optimizer. To generate the embeddings, the Neural Bag-of-Words from Subsection 3.2 has been used.

4 Caller Graph Generation

We generated caller graphs for 4.197 GitHub repositories corresponding to the CodeSearchNet Challenge's Java subset (Fig. 3 shows a small visualized subgraph). 3.787 caller graphs are used for training, 189 for testing and 221 for validation of our GCN model (see also Table 1). The distribution follows the CodeSearchNet Challenge's original split.

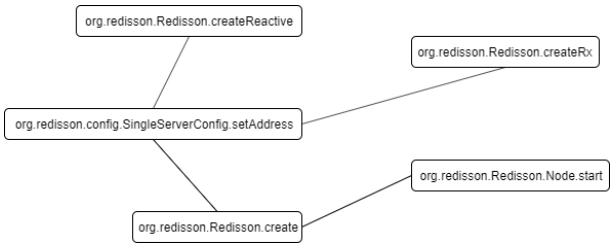

Fig. 3: Section of the caller graph generated from a repository called *redisson*. Rectangles symbolize methods and edges symbolize calls. We consider caller graphs as undirected graphs.

We reused the (comment, code) pairs provided in the CodeSearchNet Challenge's Java subset. Additionally, we extracted code fragments from the corresponding GitHub repositories to ensure that the caller graphs are as deep as possible to get the most benefit from using a GCN. Note that by doing this, we did not filter for the quality criteria used by GitHub for creating the CodeSearchNet corpus (e.g. removing test methods) [Hu19]. The additional data only serves to enhance the CodeSearchNet Challenge's methods. No additional (comment, code) pairs have been added for training.

Because the reused (comment, code) pairs are provided in a preprocessed way, the additionally extracted code fragments had to be preprocessed, too. This has been done by applying camelcase splitting, stopword filtering and lowercasing. We observed a huge amount of non-latin characters in Unicode representation in both the provided (comment, code) pairs and the additionally extracted code fragments and therefore transliterated all data into ASCII representations using *Unidecode* [7], a Python library.

4.1 Pipeline

Our caller graph generation pipeline consists of the following steps:

1. **Repository crawling** – The repositories are cloned in the version of the commit sha contained in the CodeSearchNet Challenge. (comment, code) pairs of one project are always of the same revision.
2. **Code and comment extraction** – Additional code and comments are extracted from the crawled repository using the *TreeSitter* [8] Java parser.
3. **Generate individual caller graphs** – For each method, a caller graph is generated using *Doxygen*, a software documentation tool. Each caller graph contains the fully qualified method name and its full call hierarchy, representing each referenced method as a node.
4. **Caller graph assembly** – Combining the separate caller graphs of one repository's methods results in a complete representation of its call hierarchy. Additionally, each node is enriched by the corresponding code and comment.

Finally, a vocabulary is extracted containing all code and docstring tokens over all assembled caller graphs.

Overall, we assembled 4.197 caller graphs (see Table 1) containing 1.954.908 methods. Table 2 shows the composition in more detail. It should again be noted that only the docstrings of the 213.203 methods from the CodeSearchNet Challenge have been used to train the GCN. Self-extracted docstrings have not been used.

[7] https://pypi.org/project/Unidecode/
[8] https://tree-sitter.github.io/tree-sitter/

	Train	Test	Valid	Overall
Full CodeSearchNet Dataset	4.292	239	238	4.769
Usable in GCN	3.787	189	221	4.197

Tab. 1: Table showing how many repositories are provided in the full CodeSearchNet Challenge's Java subset and how many could actually made usable for the GCN.

	Train	Test	Valid	Overall
Methods in caller graph	**1.798.287**	**64.674**	**91.947**	**1.954.908**
Methods in caller graph (provided by CodeSearchNet Challenge)	198.092	7.605	7.506	213.203
Methods in caller graph (self-extracted)	1.600.195	57.069	84.441	1.741.705
Methods in caller graph without docstring	1.182.338	44.028	70.733	1.297.099
Methods in caller graph with docstring ((comment, code) pairs)	615.949	20.646	21.214	657.809

Tab. 2: Table showing the final composition of our generated caller graphs under two aspects: Contained in the CodeSearchNet Challenge / self-extracted and without docstring / with docstring.

4.2 Problems during Caller Graph Generation

During the generation of the caller graphs using our pipeline, several problems occured in steps 2 and 3, primarily in the context of encoding processed java files. The usage of non-standard names led to invalid caller graphs that could not be used. Overall, this resulted in a loss of 305 repositories (see Table 1).

5 Experiments

Faced with the CodeSearchNet Challenge [Hu19], we conducted several experiments. The challenge is about finding the correct code function among 1000 different code snippets using a given docstring. For this purpose, we implemented different Bag-of-Words models as well as Neural Bag-of-Words models and compared them to each other. We also conducted experiments with the Graph Convolutional Network regarding the challenge. To compare the results to the CodeSearchNet Challenge baseline, we used the test det of the CodeSearchNet Corpus for the evaluation, although the GCN approach could only use it partially for the evaluation (see Table 1).

5.1 BoW vs. Neural BoW

The Bag-of-Words and Neural Bag-of-Words approaches are based on training on the train set of the CodeSearchNet Corpus [Hu19]. For experiments with the Bag-of-Words encoder a vocabulary is necessary, which has been created from the docstring tokens and code tokens of the train set. The preprocessing of the tokens is explained in section 3.1. In our experiments, we created the vocabulary only from tokens that occurred at least twice in

the corpus. Ignoring stopwords, this resulted in a vocabulary size of 87,582 for Java and 140,002 for Python. The BoW encoder thus received an MRR of 31.55% for Java and 34.67% for Python, see Table 3. To improve our results, we gave the Bag-of-Words encoder TF-IDF weighting (BoW TF-IDF), which resulted in a significant increase of 20% for Java and Python. In this case, ignoring stopwords has no longer been necessary and therefore the vocabulary size was 87,633 for Java and 140,053 for Python. At that time we already had comparable results to the Neural Bag-of-Words results of the CodeSearchNet Challenge (NBoW CSNC [Hu19]).

Another experiment was the creation of the vocabulary using Byte-Pair Encoding (BoW BPE), which in combination with TF-IDF brought an improvement of one to two percent. Here the size of the model vocabulary generated by training was set to 5000 and the tokens were encoded using the model. Afterwards, the TF-IDF algorithm was applied. With the BPE encoding of the tokens the vocabulary size for Java was 19,781 and for Python 15,706.

Our next experiment consisted of a Neural Bag-of-Words encoder with an embedding dimension of 700, a learning rate of 0.0003, a batch size of 512 and Adam optimizer. With TF-IDF weighting and the byte-pair encoded vocabulary (NBoW TF-IDF BPE), we achieved our best result with an MRR of 56.38% for Java and 62.47% for Python, an improvement of 4.98% for Java and 4.38% for Python over the currently published Neural BoW model of the CodeSearchNet Challenge [Hu19]. Nevertheless, the CodeSearchNet Challenge baseline remains best with their attention-based model (Self-Att CSNC in table 3). Yet, including TF-IDF weighting may be an interesting approach for other neural models, too.

Encoder	MRR in percentage	
Text and Code	Java	Python
SelfAtt CSNC [Hu19]	58.66	69.22
NBoW CSNC [Hu19]	51.40	58.09
BoW	31.71	34.85
BoW BPE	29.46	28.94
BoW TF-IDF	51.52	58.38
BoW TF-IDF BPE	53.83	59.43
NBoW	51.24	55.49
NBoW TF-IDF	54.59	60.94
NBoW TF-IDF BPE	**56.38**	**62.47**

Tab. 3: Mean Reciprocal Rank (MRR) on the test set of CodeSearchNet Corpus in comparison to the baseline results of the CodeSearchNet Challenge (CSNC) [Hu19]. This is the evaluation of the CSNC training task, where the models try to rank the correct code snippet among 999 distractor snippets, when the documentation comment is given as a query.

That the matching of the Neural BoW compared to BoW has become more fuzzy and therefore better can be seen more clearly in the following example: The sample with the docstring tokens "*move bytes left or right of an offset* ." has the following code tokens:

```
public void move ( final int i from , final int i position ) {
     if ( i position = = 0 ) return ;
     final int to = i from + i position ;
     final int size = i position > 0 ? buffer . length - to : buffer . length - i from ;
     system . arraycopy ( buffer , i from , buffer , to , size ) ;
}
```

Listing 1: Code tokens of a Java function that shifts bytes left or right from an offset

The tokens *left* and *right* of the docstring do not appear in the code, but instead the token *position*. With the BoW TF-IDF BPE encoder the code was ranked at position 43. This is due to the fact that the tokens *left*, *right* and *position* are evaluated in the vocabulary as completely different tokens, which have no relation to each other. Only the dot and the token *move* count as real matches. In contrast, the NBoW TF-IDF BPE encoder ranked the code at position 5, which shows that semantically similar tokens like *position*, *left* and *right* have been learned while training.

5.2 GCN

To measure the quality of the GCN-based approach, we exclusively used the Java data sets from the CodeSearchNet Challenge. A caller graph was generated from each project, according to the process from Section 4, and trained on it. The evaluation is also based

Encoder	MRR in percentage
Text and Code	Java (GCN)
BoW	29.90
BoW BPE	28.73
BoW TF-IDF	49.27
BoW TF-IDF BPE	52.46
NBoW	48.81
NBoW TF-IDF	54.49
NBoW TF-IDF BPE	**55.01**
GCN	27.15

Tab. 4: Comparison with GCN on Java test data

exclusively on the Java test data from the CodeSearchNet Challenge. The exact procedure of the evaluation and calculation of the result is described in subsection 5.1.

Table 4 shows the results of the GCN-based approach compared to the previous approach and the CodeSearchNet Challenge results. Unfortunately, at 27,15%, the GCN result is the worst score in this comparison.

6 Conclusion

In this paper, two approaches on the task of semantic code search have been presented. The first one is based on a Neural Bag-of-Words encoder. Different variations have been tried out with TF-IDF weighting and byte-pair encoding. The best model, which used TF-IDF and byte-pair encoding, achieved an MRR of 56.38% on the Java data set and an MRR of 62.47% on the Python data set. This exceeds the Neural Bag-of-Words approach of the Code-Search-Net Challenge [Hu19], but the attention based-model of the Code-Search-Net Challenge could not be exceeded so far. The other approach dealt with semantic code search based on functional dependencies. A caller graph has been created for the search in a repository. This served as input for the GCN. The usage of the GCN, however, showed no improvement.

References

[CKC] Cambronero, J.; Kim, S.; Chandra, S.: When Deep Learning Met Code Search./, pp. 964–974.

[De20] Developers, S.: Feature extraction Tfidf-Term-Weighting — scikit-learn 0.22.2 documentation, 2020, URL: https://scikit-learn.org/stable/modules/feature_extraction.html#tfidf-term-weighting.

[Hu19] Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search, 2019, arXiv: 1909.09436 [cs.LG].

[Ka19] Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K.: Pre-trained Contextual Embedding of Source Code./, pp. 1–22, 2019, arXiv: 2001.00059, URL: http://arxiv.org/abs/2001.00059.

[KW16] Kipf, T. N.; Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks, 2016, arXiv: 1609.02907 [cs.LG].

[NGV08] Narayan, G.; Gopinath, K.; Varadarajan, S.: Structure and Interpretation of Computer Programs. 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering/, June 2008, URL: http://dx.doi.org/10.1109/TASE.2008.40.

[Re09] Reiss, S. P.: Semantics-based code search. Proceedings - International Conference on Software Engineering/, pp. 243–253, 2009, ISSN: 02705257.

[Sa18] Sachdev, S. S.; Li, H. H.; Luan, S. S.; Kim, S. S.; Sen, K. K.; Chandra, S. S.: Retrieval on source code: A neural code search. MAPL 2018 - Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, co-located with PLDI 2018/, pp. 31–41, 2018.

[VM16] Vassileios Balntas Edgar Riba, D. P.; Mikolajczyk, K.: Learning local feature descriptors with triplets and shallow convolutional neural networks. In (Richard C. Wilson, E. R. H.; Smith, W. A. P., eds.): Proceedings of the British Machine Vision Conference (BMVC). BMVA Press, pp. 119.1–119.11, Sept. 2016, ISBN: 1-901725-59-6, URL: https://dx.doi.org/10.5244/C.30.119.

[Wa19] Wan, Y.; Shu, J.; Sui, Y.; Xu, G.; Zhao, Z.; Wu, J.; Yu, P.: Multi-modal attention network learning for semantic source code retrieval. Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019/, pp. 13–25, 2019.