Lightweight Semantic Code Search with TF-IDF Based Embeddings

Ted Kim*
Stanford University
tedkim97@stanford.edu

Abstract

We develop a faster approach to semantic code search - the process of retrieving relevant code from natural language queries - by using TF-IDF to convert a corpus of code snippets and queries into embeddings and using cosine similarity to retrieve the closet matches. Furthermore, we conduct other experiments with respect to preprocessing, similarity measurements, and machine-learning based embeddings to assess changes in search performance. Our results show that the model's search ability is acceptable and comparable to models like neural-bag-of-word or self-attention based embeddings, but is outclassed by SOTA models such as CodeBert, MuCoS, and CodeNN. We also show that similarity metrics and token filtering have little effect on query retrieval when trained within the codesearchnet corpus, and that document embedding models are unable to reach comparable performance in the same computational capacity.

1 Introduction

Code search is a critical practice within software engineering - programmers often search through code to understand API usage, find code purpose, or fix bugs [33, 36]. Modern search tooling involves string matching, function signatures, or log matching. While powerful, these search tools often require a prior understanding of the structure or terminology of codebases as well as learning the specifications of the tool. These details make searching for code a nontrivial learning process and lowers inexperienced users' productivity. In the past few years, there has been more research into "Semantic Code Search" - the process of retrieving relevant code from a natural language query. Natural language queries could ease these problems by allowing users to search for code in non-specific, natural language - instead of querying through the specific structure and semantics of code.

One popular subset of research is deep-learning based methods for semantic search [2, 3, 5, 6, 9, 9, 10, 15, 18, 20–22, 35, 38, 38, 39, 41].These methods are sophisticated and perform impressively, but the architectural complexity and hardware requirements make these solutions timely and costly. For instance, the reported training times for models can range from 1.6 hours to 66 hours [10, 15]. Using a price range of $3 to $25 per hour for machine learning compute[34], the cost of training a single model could range from $4 to $1500 per model.

Furthermore, these approaches ignore factors of software engineering such as the high velocity of code changes. Code in software environments can go through drastic deletions, insertions, or refactoring as new requirements, technologies, and abstractions are introduced. These factors potentially complicate the training process, as a useful code search tool might require models to be retrained on a weekly, daily, or hourly basis.

We approach semantic search through a lightweight, faster information-retrieval style and evaluate this method by measuring normalized discounted cumulative gain and accuracy on the `CodeSearchNet` corpus[15]. The input to this algorithm is a corpus of tokenized code and available documentation. We further clean the data by filtering language-specific keywords and concatenating the code and documentation token lists into a single string. Afterwards, we embedded the collection of documents into vectors using a Term-Frequency Inverse Document Frequency (TF-IDF) weighting scheme. Querying is completed by embedding natural language queries into vectors with same TF-IDF embedding, and selecting the queries with the highest cosine similarity.

1.1 Related Works

Recent literature has shown a variety of methods for semantic code-search using aspects of code property [8, 17, 23, 25, 28, 31], query refinement [13], NLP [40], or deep-learning [2, 3, 5, 6, 9, 9, 10, 15, 18, 20–22, 35, 38, 38, 39, 41, 41].

The approach in `Neural Code Search` (NCS) - creating document embeddings with `FastText` with TF-IDF weighting - seems the most intuitive and serves as the inspiration for this research[32]. Although, the purpose of additional

*tedkim97@gmail.com, nablag.com

weightings to Fast-Text embeddings are unclear, especially if document embedding are supposed to generate distinctions for different documents. A approach we disagree with is the splitting of function names into pure semantics (for instance "pxToDp" becomes the tokens "px", "to", and "dp" rather than the token "pxtodp"). This implies that chained steps or functions cannot be a unique part of codebase vocabulary such as "pxToDp" or "calculateAndAddTax".

There are several examples of code search approaches that use data augmentation with ensemble learning[5], recurrent neural networks [9], convolutional neural networks [20, 35], graph neural networks [22], transformers [6], and attentioned-based extension of NCS[3]. As mentioned earlier, these approaches perform very well, but the tradeoff in training time is non trivial - and may not be practical for real-life code search.

While there are solid foundations for evaluation methods and benchmarks[2] with adoption among other research [5, 6, 22, 39], user studies seem to be rare. Only a few papers in the literature of semantic code search conducts user studies[13, 17, 23, 32].

2 Dataset And Features

For the experiments we used the `CodeSearchNet` corpus. The corpus consists of code from publicly available, license-compatible repositories that are used by at least one other project. The columns of the dataset include the repository name, path to code snippet, url to snippet, code, tokenized code, documentation, tokenized documentation, and language. In Python, there was 1,156,085 functions with 4,481,677 unique tokens with a median number of tokens being 63 and number of documentation tokens being 0. More statistics of the corpus are in table 1.

The parameters used for training are tokenized code and tokenized documentation with the url to the snippet being used for query evaluation.

Listing 1: Example of Code from Dataset

```
def sina_xml_to_url_list(xml_data):
    """str->list
    Convert XML to URL List.
    From Biligrab.
    """
    rawurl = []
    dom = parseString(xml_data)
    for node in dom.getElementsByTagName('
    durl'):
        url = node.getElementsByTagName('
    url')[0]
    ...
```

Listing 2: Example of Preprocessed Code

```
sina_xml_to_url_list xml_data rawurl dom
    parseString xml_data node dom
    getElementsByTagName durl url node
    getElementsByTagName url rawurl append
     url childNodes data ...
```

3 Method

3.1 Preprocessing

Tokenization & Casing Code snippets were parsed by using `TreeSitter`, a parser generator tool and incremental parsing library. `TreeSitter`, generates tokens that are suited for abstract syntax trees by delimiting on language syntax (parenthesizes, commas, and more) rather then spaces). Afterwards both code and documentation tokens were converted to lowercase and english stopwords were removed. It's important to note that implementations of TF-IDF automatically lowercase and remove stop words. As a result, this step may seem redundant.

Filtering Language Syntax The code tokens include words and characters such as parenthesis, commas, and reserved keywords. These tokens have meaning attached to them such as grouping operations, changing values, applying logic, but they exist in all code snippets and can serve more as noise rather than data. As a result, we filter out these language tokens during the preprocessing step. We will later demonstrate that syntax filtering has a negligible impact on performance.

Concatentation After the code tokens have been tokenized and filtered and the documentation tokens have been lowercased, these two token lists are concatenated into a space-delimited string.

3.2 Training & Embedding

Term Frequency - Inverse Document Frequency (TF-IDF) is the method used to convert a N-sized corpus into a matrix $E_{tfidf} \in \mathbb{R}^{N \times f}$. In this context, TF-IDF is a technique converting our code snippet into an f dimensional embedding. Given a Corpus of documents, C, a document at index i, d_i, and a term within the corpus, t, a TF-IDF matrix can be computed like so[3]:

$$tfidf(t, d_i, C) = tf(t, d_i) \times (1 + \frac{|C| + 1}{1 + df(t, C)})$$
$$df(t, C) = \sum_{i}^{|C|} 1\{t \in d_i\}$$

Term Frequency (tf) is the frequency that a term (code token) appears in a document and Inverse Document Frequency (idf) represents a penalizing factor proportional to the frequency that term appear within documents. A natural interpretation of TF-IDF is an information-based weighting of each term. TF-IDF scores terms by the quantity of new information it

[2] such as CodeSearchNet[15], CosBench [39], and CodeNN[9]

[3] Other approaches use sublinear tf which is $1 + log(tf(t, d_i)$. Also note that we smooth IDF by adding 1 to the numerator and denominator.

Table 1: Dataset Code & Document Token Length Statistics

Language	Functions	Median Code	95%tile Code	Average	Median Documentation	95%tile Documentation	Average Documentation
python	1156085	63	301	104.9	0	30	7.1
go	726768	64	348	116.6	0	47.0	11.8
java	496688	65	319	110.2	0	29	6.1
javascript	1857835	83	603	261.9	0	7	1.1

introduces within a document. High term-frequencies tokens might communicate new information about a document, but high document frequency tokens indicate that the information it conveys is common.

As a result, the TF-IDF of tokens that appear frequently throughout all documents (like the keywords `return` or `def`) are weighed down by their document frequencies. By contrast, a term with a high term frequency and high document frequency will have a high TF-IDF score for those documents. A row within our TF-IDF matrix is a document represented with TF-IDF weighting applied to its tokens.

After generating a preprocessed corpus, we use a TF-IDF vectorizer to transform the corpus into embeddings. Depending on the parameters of the vectorizer, the shape of the output matrix can change. For instance, the shape of the output is $N \times 15309$ when the threshold for minimum document frequency to be 0, and decreases to $N \times 3498$ when the threshold is 0.3. The parameters of the transformer are preserved for vectorizing natural language queries. The code for TF-IDF transformation was used from scikit-learn[27]

3.3 Retrieval

Cosine Similarity Search We retrieve results from a natural language query by converting the query into a vector using the TF-IDF vectorizer trained in the previous step. As a result the query will be a sparse vector that exists in the f-dimensional space.

Afterwards, we find the vectors with the highest cosine similarity in E_{tfidf}, retrieve the index of those vectors, and return the corresponding url by these indices.

3.4 Evaluation

The evaluation metrics used were normalized discounted cumulative gain (NDCG) and URL Coverage. NDCG was chosen because it is a commonly used metric of ranking quality within information retrieval and was a key evaluation metric for `CodeSearchNet`. NDCG is a way of ranking the cumulative gain ranking across different retreival methods and is compute by:

$$NDCG_k = \frac{DCG_k}{IDCG_k} = \frac{\sum_{i=1}^{k} \frac{rel_i}{log_2(i+1)}}{\sum_{i=1}^{|REL_p|} \frac{2^{rel_i}-1}{log_2(i+1)}}$$

URL coverage was included as a metric to compare with the baselines. Coverage is calculated as the number of returned URLs for a query in a manually crafted file denoting relevancy of code snippet to query [15].

4 Experiments

4.1 RQ1: How Well Does The Algorithm Work?

Querying Our approach seems to perform well for some natural language query. For example, the query "priority queue" returns function such as:

- `pop_all` ("Non-blocking pop all in queue")
- `push` ("Push the item in the priority queue. If priority is not given, priority is set to the value of item")
- `addQUnit` (No documentation - but it appends an item to a queue)

There are weaknesses as well. The query "convert int to string" will correctly return functions like `convert_number` or `int2base36`, but it will also return results that have the same tokens ("convert", "int", "to", "string"), but the wrong semantics such as incorrect ordering ("convert string to int"):

- `datetime_from_str` ("Converts string to datetime")
- `html_to_red_green_blue` (No documentation, but this function converts an html element into a tuple of three integers)
- `convert_string` - ("Converts a string to an int or float")

Processing Time The model generates the weights much faster than the deep-learning models with an average of $\sim$10-20 minutes depending on the parameters of the vectorizer. By contrast, the models presented in `CodeSearchNet` challenge took 6-10 hours to train with similar performance, and an experimental Doc2Vec model took $\sim$4-10 hours having much worse natural language query performance.

4.2 RQ2: How Does This Compare With SOTA

Our model performs adequately with more consistency across the filtered programming languages and may beat the nbow and selfatt models reproducible, baselines MRR of 0.5809 and 0.6922 respectively[15]. However, it is much weaker than CodeBERT (MRR = 0.8685) [6], MuCoS (MRR = 0.754) [5], or DGMS (MRR = 92.2)[22]. On the other hand, the processing time to use this technique is much faster than SOTA models.

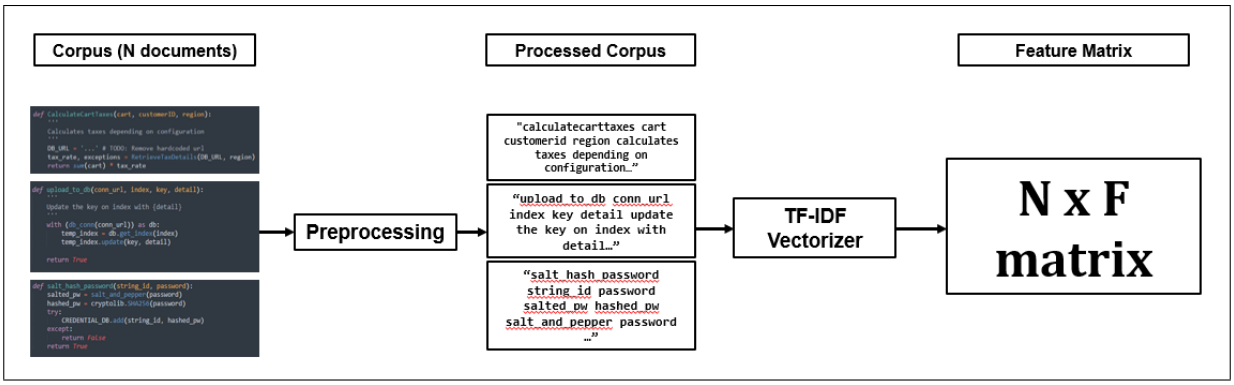
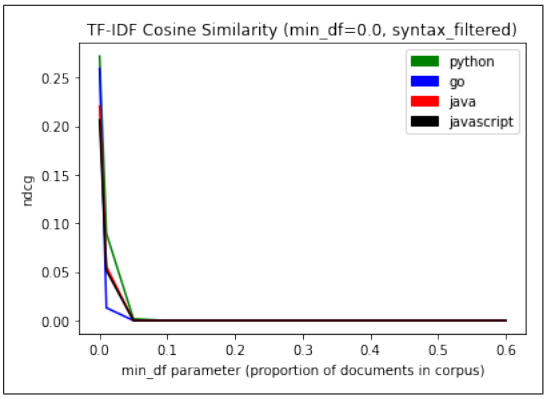


Figure 1: Diagram of the Corpus Processing Pipeline. A corpus of functions with N documents is preprocessed then it is transformed by a TF-IDF Vectorizer with different hyperparameters into a $\mathbb{R}^{n \times f}$ feature matrix. Each row represents a document converted in a f-lengthed vector

Figure 2: NDCG performance of models with different thresholds for minimum document frequency parameters.

(a) NDCG of model with corpus-proportional minimum document frequency threshold filtering



(b) NDCG of model with fixed minimum document frequency threshold filtering

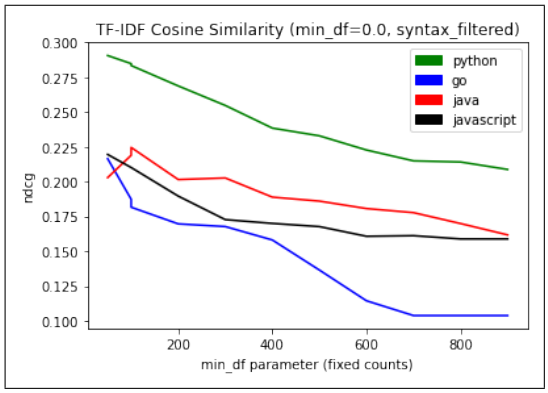


Table 2: NDCG (left) & URL Coverage % (right) Performance Comparison

Model	Python (NDCG)	Go	Java	Javascript	Python	Go	Java	Javascript
nbow	0.306	0.221	0.257	0.232	24.77	15.06	21.65	21.32
selffatt	0.239	0.101	0.173	0.153	17.09	6.63	11.19	9.40
tfidf	0.291	0.259	0.225	0.220	24.47	24.10	24.72	22.88

4.3 RQ3: How Do TF-IDF Parameters Affect Querying?

We tested the querying ability of our process by running trials with different parameters for minimum document frequency and maximum document frequency. These parameters are thresholds for including terms within the TF-IDF vocabulary.

From our results we've determined that setting a minimum document frequency too high, or a maximum document frequency too low negatively affects the performance of querying. This is a natural conclusion as setting these thresholds too high or low strays away from removing noise, and removes information. Refer to Figures 2 and 3 for more detail.

4.4 RQ4: Does Syntax Filtering Change Performance?

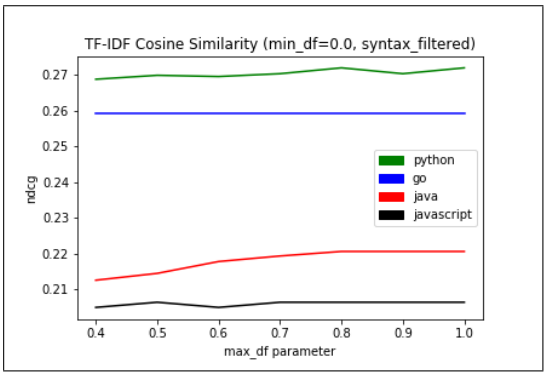
We tested the effectiveness of token filtering by running trials with different TF-IDF parameters with and without the token filtering step. Our results seem to indicate that token filtering has statistically insignificant effect on performance. More measurement are required for conclusive results.

Table 3: Similarity Metrics Differences (Python)

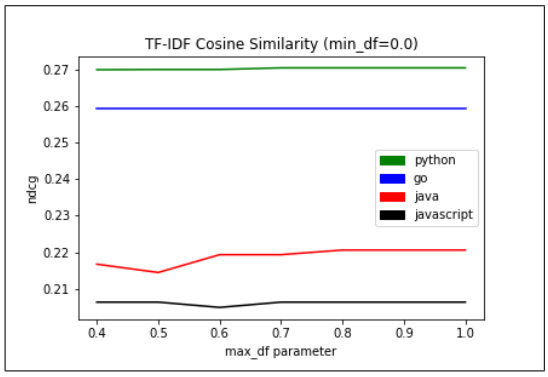
Similarity	Token Filtering	Accuracy(%)	NDCG	Accuracy (filtered)	NDCG (filtered)
cosine	none	24.47	0.285	24.27	0.283
rbf	none	24.47	0.285	15.87	0.198
linear	none	24.47	0.285	24.47	0.283

Figure 3: NDCG performance of models with different TF-IDF parameters.

(a) NDCG of model with syntax filtering



(b) NDCG of model without syntax filtering



4.5 RQ5: How Do Similarity Measurements Affect Search?

We experimented with using other similarity measures such as kernels, like the RBF kernel and linear kernel with Python. We set the threshold for minimum document frequency to 100 words per our prior results with no threshold for maximum document frequency as these parameters tend to give the highest performance. As we see in table 3, the RBF kernel seems to be sensitive to the exclusion of filtering, unlike the cosine similarity. The results for cosine and linear similarity metrics were identical because the TF-IDF vectorizer outputs normalized vectors - which causes the cosine similarity to be identical as the linear kernel.

4.6 RQ6: Would Performance Improve with a Machine-Learning-Based Embedding?

We attempted to use `Doc2Vec`, an embedding technique for documents, similar to `Word2Vec`, using `Gensim` [30]. `Doc2Vec` was chosen because it's a learning algorithm designed for generating document embeddings that avoids the ordering weaknesses we observed from using TF-IDF.

We experimented with different hyperparameters such as "vector size", "word window", and "training epochs"; however, we were unable to train a successful model within our computational limits. Our best possible model used a vector size of 300, a word window of 15 and was trained for 30 epochs and it achieved a NDCG of 0.131.

A possible explanation is that document embedding techniques like `Doc2Vec` have relatively poor performance with short-length documents, such as code tokens[4]. Another possible reason for the poor performance is that the corpus is small compared to successful use cases of `Doc2Vec`.

5 Conclusion & Discussion

In conclusion, we attempt a faster semantic code search with TF-IDF vectors and cosine similarity search. Although this method's code-retrieval ability struggles with some basic semantic hurdles such as sequencing of words, it has acceptable performance compared to more sophisticated models like NBoW and SelfAttention representation while being much faster to train - demonstrating a feasibility with rapidly changing code bases. In addition, we discovered that similarity metrics and token filtering have little effect on query retrieval when trained within a language corpus.

In the future, we would like to perform additional experiments to understand how documentation and code contribute to the query retrieval. Furthermore, we would like to improve the performance of our approach. One possible avenue is to augment the data by applying the function naming deconstruction (converting "pxToDp" to "px", "to", "dp")[32] in addition to preserving the original terminology "pxToDp". Another approach is to use the typing and data flow within functions to modify the vectorization of the natural language query, similar to the approach in [28]. Another avenue worth pursuing is to use `word2vec` models to generate feature vectors and use them for similarity search. Finally, we may be able to pursue pre-trained Doc2Vec models to convert code snippets into vectors to have better embeddings.

Aside from model performance, we would like to increase the quality of evaluation by using more common practices as well as conducting user studies. The first involves adopting Mean Reciprocal Rank as well as incorporating other benchmarks such as CosBench and CodeNN datasets, and the latter would involve running experiments and user studies comparing this approach to pre-establishing codesearch tools for a usability evaluation reference.

References

[1] Ming Li A, Hang Li B, and Zhi-hua Zhou A. Contents lists available at ScienceDirect Information Processing and Management.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *arXiv:1709.06182 [cs]*, May 2018. arXiv: 1709.06182.

[3] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 964–974, New York, NY, USA, August 2019. Association for Computing Machinery.

[4] Cedric De Boom, Steven Van Canneyt, Steven Bohez, Thomas Demeester, and Bart Dhoedt. Learning semantic similarity for very short texts. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 1229–1234, 2015.

[5] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning for Semantic Code Search. *arXiv:2107.04773 [cs]*, July 2021. arXiv: 2107.04773.

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv:2002.08155 [cs]*, September 2020. arXiv: 2002.08155.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[8] Isabel Garcia-Contreras, Jose F. Morales, and Manuel V. Hermenegildo. Semantic Code Browsing. *arXiv:1608.02565 [cs]*, August 2016. arXiv: 1608.02565.

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944, May 2018. ISSN: 1558-1225.

[10] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. A Multi-Perspective Architecture for Semantic Code Search. *arXiv:2005.06980 [cs]*, May 2020. arXiv: 2005.06980.

[11] Arash Heidarian and Michael J. Dinneen. A Hybrid Geometric Approach for Measuring Similarity Level Among Documents and Document Clustering. In *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 142–151, March 2016.

[12] Arash Heidarian and Michael J. Dinneen. A hybrid geometric approach for measuring similarity level among documents and document clustering. In *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 142–151, 2016.

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

[14] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[15] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[16] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, November 2015.

[17] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 1358–1363, Atlanta, Georgia, July 2010. AAAI Press.

[18] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 946–957, New York, NY, USA, May 2018. Association for Computing Machinery.

[19] Quoc V. Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. *arXiv:1405.4053 [cs]*, May 2014. arXiv: 1405.4053.

[20] Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. Learning Code-Query Interaction for Enhancing Code Searches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126, September 2020. ISSN: 2576-3148.

[21] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. Adaptive Deep Code Search. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 48–59, New York, NY, USA, July 2020. Association for Computing Machinery.

[22] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. Deep Graph Matching and Searching for Semantic Code Retrieval. *ACM Transactions on Knowledge Discovery from Data*, 15(5):88:1–88:21, May 2021.

[23] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):152:1–152:28, October 2019.

[24] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[25] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, October 2012. Association for Computing Machinery.

[26] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1066–1082, New York, NY, USA, June 2020. Association for Computing Machinery.

[29] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[30] Radim Rehurek and Petr Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[31] Steven P. Reiss. Semantics-based code search. In *2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, May 2009. ISSN: 1558-1225.

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

[33] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA, August 2015. Association for Computing Machinery.

[34] Amazon Web Services. Ec2 on-demand instance pricing – amazon web services. *Amazon Web Services, Inc.*

[35] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. Improving Code Search with Co-Attentive Representation Learning. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 196–207, New York, NY, USA, July 2020. Association for Computing Machinery.

[36] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An Examination of Software Engineering Work Practices, 1997.

[37] Laurens van der Maaten and Geoffrey Hinton. Visualizing non-metric similarities in multiple maps. *Machine Learning*, 87(1):33–55, April 2012.

[38] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, August 2016. Association for Computing Machinery.

[39] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 344–354, 2020.

[40] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning. In *Proceedings of The Web Conference 2020*, WWW '20, pages 2309–2319, New York, NY, USA, April 2020. Association for Computing Machinery.

[41] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. OCoR: An Overlapping-Aware Code Retriever. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 883–894, September 2020. ISSN: 2643-1572.