Bag-of-Words Baselines for Semantic Code Search

Xinyu Zhang,[1] **Ji Xin,**[1] **Andrew Yates,**[2] and **Jimmy Lin**[1]

[1] David R. Cheriton School of Computer Science, University of Waterloo
[2] Max Planck Institute for Informatics, Saarland Informatics Campus

Abstract

The task of *semantic code search* is to retrieve code snippets from a source code corpus based on an information need expressed in natural language. The semantic gap between natural language and programming languages has for long been regarded as one of the most significant obstacles to the effectiveness of keyword-based information retrieval (IR) methods. It is a common assumption that "traditional" bag-of-words IR methods are poorly suited for semantic code search: our work empirically investigates this assumption. Specifically, we examine the effectiveness of two traditional IR methods, namely BM25 and RM3, on the CodeSearchNet Corpus, which consists of natural language queries paired with relevant code snippets. We find that the two keyword-based methods outperform several pre-BERT neural models. We also compare several code-specific data pre-processing strategies and find that specialized tokenization improves effectiveness. Code for reproducing our experiments is available at https://github.com/crystina-z/CodeSearchNet-baseline.

1 Introduction

Community Question Answering forums like Stack Overflow have become popular[1] methods for finding code snippets relevant to natural language questions (e.g., *"How can I download a paper from arXiv in Python?"*). Such forums require community members to provide answers, which means that potential questions are limited to public code, and a large portion of questions cannot be answered in real time. The task of semantic code search removes these limitations by treating a code-related natural language question as a query and using it to retrieve relevant code snippets. In this way, novel questions can be immediately answered whether in public or private code repositories.

Consequently, the semantic code search task is receiving an increasing amount of attention. Several early efforts showed promising results applying neural networks models to various code search datasets (Gu et al., 2018; Sachdev et al., 2018; Cambronero et al., 2019; Zhu et al., 2020; Srinivas et al., 2020). To facilitate research on semantic code search, GitHub released the CodeSearchNet Corpus and Challenge (Husain et al., 2019), providing a large-scale dataset across multiple programming languages with unified evaluation criteria. This dataset has been utilized by multiple recent papers (Feng et al., 2020; Gu et al., 2021; Sun et al., 2020; Arumugam, 2020).

Work on semantic code search has focused on neural ranking models under the assumption that such methods are necessary to bridge the semantic gap between natural language queries and relevant results (i.e., code snippets). Such approaches usually design a task-specific joint vector representation to map natural language queries and programming language "documents" into a shared vector space (Gu et al., 2018; Sachdev et al., 2018; Cambronero et al., 2019). Inspired by progress in pre-trained models (Devlin et al., 2019), researchers proposed CodeBERT (Feng et al., 2020), a pre-trained transformer model specifically for programming languages, which yields impressive effectiveness on this task.

Beyond utilizing the raw text of code corpora, another thread of research conducts retrieval using structural features parsed from code, which are believed to contain rich semantic information (Srinivas et al., 2020). Multiple papers have also proposed incorporating structural information with neural ranking models (Gu et al., 2021; Sun et al., 2020; Ling et al., 2021; Guo et al., 2020).

[1] https://stackoverflow.blog/2020/01/21/scripting-the-future-of-stack-2020-plans-vision/

Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pages 88–94

In contrast to these comparatively sophisticated methods, in this work we explore the effectiveness of traditional information retrieval (IR) methods on the semantic code search task. This exploration is of interest for two reasons:

First, while neural methods can take advantage of distributed representations (i.e., static or contextual embeddings) to model semantic similarity, Yang et al. (2019) found that pre-BERT neural ranking models can underperform traditional IR methods like BM25 with RM3 query expansion, especially in the absence of large amounts of data for training. Prior work has claimed that traditional IR methods are unfit for code search (Husain et al., 2019), but there is a lack of empirical evidence supporting this claim. In fact, in one of the few comparisons with traditional IR methods available (Sachdev et al., 2018), BM25 performed well in comparison to the proposed neural methods on an Android-specific dataset.

Second, neural approaches are often reranking methods that rerank candidate documents identified by a first-stage ranking method. Even dense retrieval methods that perform ranking on shared vector representations directly can benefit from hybrid combinations with keyword-based signals as well as another round of reranking (Gao et al., 2020). It is thus useful to identify the best-performing traditional IR methods in this domain, so that they can provide a complementary source of evidence.

Thus, our work has two main contributions: First, we provide strong keyword baselines for semantic code search, demonstrating that traditional IR methods can in fact outperform several pre-BERT neural ranking models even without a semantic matching ability, which extends the conclusions drawn by Yang et al. (2019) on *ad hoc* retrieval to the semantic code search task. Second, we investigate and quantify the impact of specialized pre-processing for code search.

2 Related Work

As discussed above, joint-vector representations have been widely used in recent work on code search. NCS (Sachdev et al., 2018) proposed an approach integrating TF-IDF, word embeddings, and an efficient embedding search technique where the word embeddings are learned in an unsupervised manner. CODEnn (Gu et al., 2018) developed a neural model based on queries and separate code components. UNIF (Cambronero et al., 2019) investigated the necessity of supervision and sophisticated architectures for learning aligned vector representations. After concluding that supervision and a simpler network architecture are beneficial, the authors further enhanced NCS by adding a supervision module on top. In addition to introducing the dataset, the CodeSearchNet paper also proposed joint-embedding models as baselines, where the embeddings may be learned from neural bag of words (NBoW), bidirectional RNN, 1D CNN, or self-attention (SelfAtt). In this work, we compare against the best-performing of these baselines, NBoW and SelfAtt.

Unlike attempts to learn aligned vector representations from each dataset, CodeBERT (Feng et al., 2020) built a BERT-style pre-trained transformer encoder with code-specific training data and objectives, and then fine-tuned the model on downstream tasks. This approach has been highly successful.

Another line of work tries to enhance retrieval by incorporating structural information. In work where queries and code snippets are encoded separately, this is usually achieved by merging the encoded structure into the code vector. Sun et al. (2020) extracted paths from the abstract syntax tree (AST) of the code and directly used the encoded path to represent the code snippet. Gu et al. (2021) built a statement dependency matrix from the code and transformed it into a vector, which is then added to the code vector prepared from the text. Ling et al. (2021) utilized a graph neural network to embed the *program graph* into the code vector. Adopting a different approach, Guo et al. (2020) extended CodeBERT by adding two structure-aware pre-training objectives, and showed that the benefits of structural information are orthogonal to the benefits of large-scale pre-training.

While neural ranking models are popular approaches to the code retrieval task, we found few papers that compared them with traditional algorithms. To the best of our knowledge, only Sachdev et al. (2018) compared their embedding model with BM25, finding that BM25 performed acceptably.

3 Models

In this section, we describe the traditional IR methods that we used in our experiments and the neural ranking models that have been evaluated on the CodeSearchNet Corpus in previous work (Husain et al., 2019; Feng et al., 2020).

3.1 Traditional IR Baselines

To test the effectiveness of traditional IR methods, we chose two well-known and effective retrieval methods as our baselines: BM25 (Robertson and Zaragoza, 2009) and RM3 (Lavrenko and Croft, 2001; Abdul-Jaleel et al., 2004). Both have been widely used for *ad hoc* retrieval and have been demonstrated to be strong baselines compared to multiple pre-BERT neural ranking models (Yang et al., 2019).

BM25 is a ranking method based on the probabilistic relevance model (Robertson and Jones, 1976), which combines term frequency (tf) and inverse document frequency (idf) signals from individual query terms to estimate query–document relevance. RM3 is a query expansion technique based on pseudo relevance feedback (PRF) that can be combined with another ranking method such as BM25. It expands the original query with selected terms from initial retrieval results (e.g., results of BM25) and applies another round of retrieval (e.g., with BM25) using the expanded query. We omit a comprehensive explanation of these two methods here and refer interested readers to the cited papers.

3.2 Neural Ranking Models

We compare the traditional IR methods described above with three neural ranking models: neural bag of words (NBoW), self-attention (SelfAtt), and CodeBERT. Results of the first two models are reported by Husain et al. (2019), and the last model by Feng et al. (2020). We use their reported scores in this paper.

According to Husain et al. (2019), both NBoW and SelfAtt encode natural language queries and code into a joint vector space, and then aggregate the sequence representation into a single vector. The models are trained with the objective of maximizing the inner products of the aggregated query vectors and code vectors. The two models only differ in the *encoding* step, where NBoW encodes each token through a simple embedding matrix and SelfAtt encodes the sequence using BERT (Devlin et al., 2019). Feng et al. (2020) pre-trained a bi-modal (natural language and programming language) transformer encoder based on RoBERTa (Liu et al., 2019), with the hybrid objectives of Mask Language Model (MLM) and Replaced Token Detection (RTD). The model is then fine-tuned for the code search task on each programming language dataset. We refer readers to the original papers (Husain et al., 2019; Feng et al., 2020) for further model details and hyper-parameters.

	Datapoints	Unique Docstrings			
		Total	*Training*	*Validation*	*Test*
Go	346 365	277 118	253 979	11 757	11 382
Java	496 688	372 894	340 380	11 621	20 893
JS	138 625	123 738	111 443	6 876	5 419
PHP	578 118	424 657	387 470	17 843	19 344
Python	457 461	421 263	379 864	20 897	20 502
Ruby	53 279	47 763	43 549	2 089	2 125
All	2 070 536	1 667 433	1 516 685	71 083	79 665

Table 1: Dataset Size Statistics.

4 Dataset and Pre-processing

In this section, we introduce the CodeSearchNet Dataset (Husain et al., 2019) used in this paper and the code specific pre-processing strategies (e.g., tokenization) to be compared.

4.1 Dataset

CodeSearchNet[2] is a proxy dataset prepared from non-fork open-source Github repositories. It consists of 2M docstring–code pairs and 4M unlabeled code fragments, where the code fragments are function-level snippets and their respective docstrings (if any) serve as substitutes for natural language queries. Under CodeSearchNet, there are two sub-datasets, namely CodeSearchNet Corpus and CodeSearchNet Challenge. The CodeSearchNet Corpus dataset uses 2M docstrings as automatically-labeled queries, whereas the CodeSearchNet Challenge dataset uses another 99 free-text queries that were manually judged.

In this work we conduct all experiments on the CodeSearchNet Corpus dataset. The labeled data are split into training, validation, and test sets in a ratio of 80:10:10. Table 1 shows the overall dataset size and the number of unique docstrings in each data split. The test set is partitioned into segments of size 1000 at the evaluation stage, and the correct code snippet for a given query is compared against the other snippets within the same segment. That is, the code snippets in the 1000 `<docstring, code snippet>` pairs naturally form the *distractor* set for each other.

4.2 De-duplication

According to Husain et al. (2019), the crawled data are filtered according to certain heuristic rules,

[2] https://github.com/github/CodeSearchNet

```ruby
# Appends the given string at the end of
    the current string value for key k.
def putcat (k, v)
  k = k.to_s; v = v.to_s
  lib.abs_putcat(@db, k, Rufus::Tokyo.
    blen(k), v, Rufus::Tokyo.blen(v))
end

# Appends the given string at the end of
    the current string value for key k.
def putcat (k, v)
  k = k.to_s; v = v.to_s
  @db.putcat(k, v)
end
```

Figure 1: Docstring duplication example (unused docstring and extra blank lines are removed).

	Docstrings	Duplicates (Number / Fraction)			
		Total		*Same repo*	
Go	277 118	18 531	6.7%	15 255	82.3%
Java	372 894	39 067	10.5%	34 072	87.2%
JS	123 738	7 254	5.9%	3 342	46.1%
PHP	424 657	42 058	9.9%	23 515	55.9%
Python	421 263	20 776	4.9%	16 608	79.9%
Ruby	47 763	3 006	6.3%	2 499	83.1%
All	1 667 433	130 692	7.4%	95 291	72.9%

Table 2: Docstring Duplicate Statistics. The *Docstring* column is the same as the the *Unique Docstring* in Table 1. *Total* indicates the number and proportion of duplicate docstrings in each programming language across the *entire* dataset. *From Same Repo* indicates the number and proportion of docstrings whose duplicates are found *all* in the same repository.

including removing (1) pairs where the docstring is shorter than three tokens, (2) functions that contain fewer than three lines, contain the "test" substring, or serve as constructors or standard extension methods, and (3) duplicate functions. Nevertheless, even though duplicate functions are removed, queries prepared from docstrings can still repeat. That is, different functions can share the same documentation. Such duplication may result from function overloading, oversimplified documentation, or mere coincidence. An example of this duplication is shown in Figure 1.

Table 2 shows that such query duplication can be observed in all programming languages to some degree, and most of the duplication arises from functions in the same repository. Considering the number of duplicate docstrings, it is inaccurate to consider all functions other than the one matched to the current query as negative samples. In this work, we aggregate all functions sharing the same docstring and regard all of them as relevant results.

4.3 Pre-processing

In all experiments, we apply the Porter stemmer and perform stopword removal using the default stopwords list in the Anserini toolkit (Yang et al., 2017), which is a Lucene-based IR system.

On top of this default configuration, we investigate the effectiveness of the following tokenization and stopword removal strategies specific to programming languages:

- no-code-tokenization: No extra pre-processing is applied other than Porter stemmer and removal of English stopwords.
- code-tokenization: Tokens in both `camelCase` and `snake_case` in code snippets and documentation are further tokenized into separate tokens, e.g., `camel case` and `snake case`.[3]
- code-tokenization + remove reserved tokens: Considering that reserved tokens in programming languages intuitively add little value in exact match methods, we remove the reserved tokens of each programming language on top of the code-tokenization condition.

We show length and vocabulary statistics after applying each pre-processing strategy in Table 3. In the table, *total vocab size* is the number of tokens that appear in either docstring or code, and *overlapped vocabulary ratio* is the percentage of tokens appearing in both docstring and code in the entire vocabulary. The table shows that code tokenization greatly shrinks the vocabulary size and raises the overlapped vocabulary ratio. Interestingly, reserved token removal shortens the code snippets length, but shows little impact on the overall vocabulary size. This results from the fact that reserved tokens are commonly contained in variable names as subtokens and thus reappear after code tokenization (e.g., the variable name `class_dir` would be tokenized into `class` and `dir`, therefore `class` would still appear in the final vocabulary).

5 Experiments

5.1 Experimental Setup

All our experiments were conducted with Capreolus (Yates et al., 2020), an IR toolkit integrating ranking and reranking tasks under the same data

[3] According to Husain et al. (2019), NBoW and SelfAtt tokenize 'camelCase' tokens into subtokens ('camel' and 'case'), which is similar to our code-tokenization setting.

		Ruby	JS	Go	Python	Java	PHP
no-code-tokenization keep reserved tokens	**avg docstring length**	14	13	20	14	15	8
	avg code length	37	73	44	60	47	45
	total vocab size	160 175	455 771	651 143	1 255 725	1 248 229	1 061 762
	overlapped vocab %	13.58%	10.18%	33.02%	9.92%	8.02%	7.49%
code-tokenization keep reserved tokens	**avg docstring len**	15	13	24	14	16	8
	avg code len	57	110	64	88	85	72
	total vocab size	31 999	76 305	68 877	191 608	105 005	134 729
	overlapped vocab %	40.13%	28.24%	36.83%	26.41%	28.62%	21.34%
code-tokenization remove reserved tokens	**avg docstring len**	15	13	28	14	16	8
	avg code len	48	93	57	78	74	62
	total vocab size	31 999	76 305	68 877	191 608	105 004	134 729
	overlapped vocab %	40.12%	28.24%	36.83%	26.41%	28.62%	21.34%

Table 3: Average length and vocabulary statistics after applying each pre-processing strategy.

Models		**Ruby**	**JS**	**Go**	**Python**	**Java**	**PHP**
CodeBERT		0.6926	0.7059	0.8400	0.8685	0.7484	0.7062
NBoW		0.4285	0.4607	0.6409	0.5809	0.4835	0.5181
SelfAtt		0.3651	0.4506	0.6809	**0.6922**	0.5866	**0.6011**
no-code-tokenization + keep reserved tokens	BM25	0.4484	0.4097	0.6979	0.4317	0.4002	0.3758
	BM25+RM3	0.4427	0.4123	0.6761	0.4216	0.3988	0.4062
code-tokenization + keep reserved tokens	BM25	**0.5789**	**0.5522**	0.7289	0.5989	**0.6022**	0.5929
	BM25+RM3	0.5735	0.5312	0.7214	0.5865	0.5777	0.5379
code-tokenization + remove reserved tokens	BM25	0.5707	0.5312	**0.7317**	0.5905	0.5838	0.5399
	BM25+RM3	0.5703	0.5269	0.7246	0.5871	0.5794	0.5400

Table 4: MRR on the test set of the CodeSearchNet Corpus where each model searches for the correct code snippet against the 999 distractors. The highest scores among non-BERT models are highlighted in **bold**, and the ones among keyword-only models are underlined. We copied the scores of neural ranking models from Husain et al. (2019) and Feng et al. (2020).

processing pipeline. We chose the toolkit to enhance reproducibility and to support future comparisons. Note that although Capreolus is primarily designed for text ranking with neural ranking models, in this work we do not use any of those features. The underlying implementation of BM25 and RM3 are provided by the Pyserini toolkit (Lin et al., 2021), which in turn is built on the Lucene open-source search library, but Capreolus provides simplified mechanisms for parameter tuning and other useful features for end-to-end experiments.

Following the original paper (Husain et al., 2019), each correct code snippet was searched against a fixed set of 999 *distractors*, as described in Section 4.1. All experiments were evaluated with Mean Reciprocal Rank (MRR). In all experiments, we tuned the parameters `k1` and `b` for BM25 and `originalQueryWeight`, `fbDocs`, `fbTerms` for RM3 on the validation set, then applied the parameters from the best result on the test set. Note that since BM25 and RM3 only require parameter tuning, we did not use the training set mentioned in Table 1.

`k1`	[0.7, 1.3], step size 0.1
`b`	[0.7, 1.0], step size 0.1
`fbDocs`	[55, 95], step size 10
`fbTerms`	2, 5, 7, 10
`originalQueryWeight`	0.7, 0.8, 0.9

Table 5: BM25 and RM3 parameter values explored.

After pilot experiments on the Ruby and Go datasets to determine reasonable parameter ranges to search, we performed a grid search on each language dataset over the values shown in Table 5.

5.2 Results and Analysis

The results are shown in Table 4. The first row reports the results of CodeBERT (Feng et al., 2020). We list this result here to better compare the IR baselines with the state-of-the-art model in the field. The next two rows are pre-BERT neural model results copied from Husain et al. (2019). The remaining rows show the scores of BM25 and RM3 with the three aforementioned pre-processing strategies on the six programming language datasets.

As Table 4 shows, BM25 and BM25 + RM3 in general outperform the NBoW and SelfAtt baselines despite variations in effectiveness across programming languages. The SelfAtt model only shows sizeable improvement over BM25 on Python and a modest improvement on PHP. This suggests that the gap between natural language and programming languages does not necessarily hinder traditional IR methods in the code search task, and that distributed representations are not necessarily better at addressing this gap.

Comparing the results of BM25 and BM25 + RM3, we observe that adding RM3, which is generally considered more effective, does not improve over BM25 on any of the language datasets. We suspect the cause of this unanticipated result is that most of the queries in CodeSearchNet only have a single relevant document, which may not be sufficient to quantify the benefits of pseudo relevance feedback techniques. This hypothesis is supported by a similar observation that adding RM3 degrades effectiveness on the MS MARCO dataset (Bajaj et al., 2018), where each query also has few relevant documents (Lin et al., 2020).

The results from each pre-processing strategy show the necessity of code tokenization, which improves MRR overall. On the other hand, removing the reserved tokens does not improve effectiveness. The possible reasons could be that (1) some reserved tokens are in the English stopwords list and would be removed anyway (e.g. `for`, `if`, `or`, etc.), (2) some special reserved tokens rarely appear in the query and thus contribute little to the final score (e.g. `elif`, `await`, etc.), and (3) frequently-appearing reserved words are given small IDF weights in BM25, which minimizes their effect (e.g. `final`, `return`, `var`).

6 Conclusion

In this paper we examined the effectiveness of traditional IR methods for semantic code search and found that while these exact match methods are not as effective as CodeBERT, they generally outperform pre-BERT neural models. We also compare the effect of code-specific tokenization strategies, showing that while splitting camel and snake case is beneficial, removing reserved tokens does not necessarily help keyword-based methods.

There are also aspects of semantic code search that this paper does not cover. Sachdev et al. (2018) mentioned the nuance between different code components, such as how readability can differ for function names and local variables. We leave for future work an investigation of whether treating such components differently improves effectiveness. Nevertheless, the lesson from our work seems clear: even with advances in neural approaches, we shouldn't neglect comparisons to and contributions from strong keyword-based IR methods.

Acknowledgements

This research has been supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

Nasreen Abdul-Jaleel, James Allan, W. Bruce Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Donald Metzler, Mark D. Smucker, Trevor Strohman, Howard Turtle, and Courtney Wade. 2004. UMass at TREC 2004: Novelty and HARD. In *Proceedings of the Thirteenth Text REtrieval Conference (TREC 2004)*, Gaithersburg, Maryland.

Lakshmanan Arumugam. 2020. Semantic code search using Code2Vec: A bag-of-paths model. Master's thesis, University of Waterloo.

Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2018. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv:1611.09268v3*.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. 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, page 964–974, New York, NY, USA. Association for Computing Machinery.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association*

for Computational Linguistics: EMNLP 2020, pages 1536–1547, Online. Association for Computational Linguistics.

Luyu Gao, Zhuyun Dai, Zhen Fan, and Jamie Callan. 2020. Complementing lexical retrieval with semantic residual embedding. *arXiv:2004.13969*.

Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. CRaDLe: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA. Association for Computing Machinery.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, L. Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2020. GraphCodeBERT: Pre-training code representations with data flow. *arXiv:2009.08366*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the state of semantic code search. *arXiv:1909.09436*.

Victor Lavrenko and W. Bruce Croft. 2001. Relevance based language models. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, page 120–127, New York, NY, USA. Association for Computing Machinery.

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'21, New York, NY, USA. Association for Computing Machinery.

Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2020. Pretrained transformers for text ranking: BERT and beyond. *arXiv:2010.06467*.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Transactions on Knowledge Discovery from Data*, 15(5):Article No. 88.

Y. Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, M. Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv:1907.11692*.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundation and Trends in Information Retrieval*, 3(4):333–389.

Stephen E. Robertson and Karen Sparck Jones. 1976. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146.

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, page 31–41, New York, NY, USA. Association for Computing Machinery.

Kavitha Srinivas, I. Abdelaziz, Julian T. Dolby, and J. McCusker. 2020. Graph4Code: A machine interpretable knowledge graph for code. *arXiv:2002.09440*.

Zhensu Sun, Y. Liu, Chen Yang, and Yu Qian. 2020. PSCS: A path-based neural model for semantic code search. *arXiv:2008.03042*.

Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the use of Lucene for information retrieval research. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, page 1253–1256, New York, NY, USA. Association for Computing Machinery.

Wei Yang, Kuang Lu, Peilin Yang, and Jimmy Lin. 2019. Critically examining the "neural hype": Weak baselines and the additivity of effectiveness gains from neural ranking models. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, page 1129–1132, New York, NY, USA. Association for Computing Machinery.

Andrew Yates, Kevin Martin Jose, Xinyu Zhang, and Jimmy Lin. 2020. Flexible IR pipelines with Capreolus. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 3181–3188, New York, NY, USA. Association for Computing Machinery.

Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An overlapping-aware code retriever. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 883–894, New York, NY, USA. Association for Computing Machinery.