ATHENA: TOWARDS IMPROVING SEMANTIC CODE SEARCH WITH KNOWLEDGE GRAPHS

Nathan Cooper
Advisor: Denys Poshyvanyk
William & Mary

April 21, 2022

Abstract

One main idea in software engineering is to reuse existing software instead of building due to the high cost. However, finding existing software that accomplishes a specific use case can be difficult. Code search has attempted to help this by allowing keyword searching for code similar to earlier search engines. Recent work has attempted to leverage the semantics present in the query and code.[3,6,12,15,16,20,29] However, these approaches do not consider the context of methods with the rest of the software. This context can be important when performing a search for determining the relevance of a returned result.[16] To address the current problems in code search we designed Athena, a semantic code search engine that leverages software's context through a knowledge graph and a graph neural network. To evaluate Athena, we used the python language portion of the popular CodeSearchNet Challenge[16] that contains a large set of query-code pairs with the relevance of the returned code to the query. We found that Athena still lags behind previous techniques that only work at the method level without additional context. Specifically, out of 2,098 repositories, Athena achieves an Normalized Discounted Cumulative Gain (NDCG) score of 0.757 with the most recent state of art achieving 0.999. We investigate the reasons behind this and discuss future work to overcome them.

Introduction

Software has become pervasive in our society, from running on our smartphones to controlling propulsion and control systems in spacecrafts. Without software, none of the current scientific revolutions would be possible. Therefore, it is important to make sure high quality software is available to everyone to ensure the progress of innovation.

The sharing of advances in software has been growing in popularity with big shifts in software companies becoming more open sourced. This has allowed for everyone to use their software as well as helping to contribute their time to improving the software. However, searching for software to reuse is still challenging in today's age.

To tackle this challenge, we introduce the current implementation of Athena, a tool for improving the relevancy of retrieved methods from a given query method using a software knowledge graph and graph neural network by capturing the context methods that are being searched. Context is important because similar to natural language with J.R. Firth's famous quote "[y]ou shall know a word by the company it keeps",[8] looking at a method's context, *i.e.,* the file, package, and project it resides in, can give a deeper insight as to its intent and functionality.[16] Specifically, we leveraged a software project's call graph to construct our software knowledge graph and used GraphSage,[14] a recently popular graph neural network architecture that we couple with a pretrained Transformer[32] neural network for encoding each method to generate the node embeddings in the knowledge graph.

We evaluated Athena on the python portion of the popular CodeSearchNet Challenge[16] benchmark where the task is to retrieve the most relevant methods to a given query such as "how to sort a list of numbers in ascending order?" Since CodeSearchNet originally only contained query-method pairs, we modified the challenge to also contain the context of each of the methods to work with Athena. Through our evaluation of 2,098 query-method pairs, we found Athena performs worse than the most recent previous state of the art approach that only uses a single method without any additional context. Lastly, we investigate the reason behind this and discuss future work to improve Athena.

In summary, this paper discusses the following:

1. Athena, a novel approach to code search that incorporates the context of methods for determining their relevancy to a user's query;

2. An evaluation of Athena compared to baselines on the popular CodeSearchNet challenge;
3. A modified version of the CodeSearchNet Challenge that allows for evaluating approaches that consider the context of methods for the task of code search;
4. And an analysis of the reasons for Athena not achieving improved results over a previous approach and a discussion of future work to improve Athena.

Implementation

Athena *Overview*

Athena takes advantage of the rich information present in the actual code and structural information present in software projects' call graphs by combining them in a non-linear way using a Graph Neural Network (GNN) algorithm. This combined information is then used for calculating the similarity between a user's query and methods within and across software systems for ranking the relevancy of the methods.

Athena has three components, namely *Neural-based Representation*, *Structural Representation*, and *Representation Fusing*, as shown in Figure 1. Each component is explained in the following subsections.

Neural-based Representation

In Athena's *Neural-based Representation* component, each method, M, in a target software project, P, is transformed into a distributed vector representation using some function f_{neural}. For this *Neural-based Representation* function, $f_{neural}(M) = V_M, M \in P$, we used a sentence-transformer[28] model that was trained on the CodeSearchNet dataset using a contrastive loss[27] so that methods and their corresponding docstrings are similar in vector space. Specifically, the text in the method and docstring is converted into a sequence of tokens using the Byte Pair Encoding algorithm[31] and converted into a continuous distributed vector. These vectors are transformed through the layers of the network into their final representation and are then averaged to produce a single vector that represents the entire method or docstring. The cosine similarity metric is used among the methods and docstrings in a batch and matching method a docstring cosine similarities are pushed to be as high as possible while mismatched methods and docstring cosine similarities are pushed down via backpropagation.

Structural Representation

In order to capture the structural aspect of a software system, we construct a knowledge graph by using the project's method-level call graph to serve as our structural representation. More formally, let $G = (N, E)$ be a target software project graph, where N represents the list of methods in the project as nodes and E represents the list of method calls between the methods in the project. Additionally, in order to capture the natural language information present in software systems, *i.e.,* the documentation of methods, we add documentation nodes and associated edges to G. Lastly, we add edges between methods that reside in the same file regardless of if they call each other as methods inside the same file usually implies that they are related in some way. In Sec. 3 we perform an ablation study of our design decisions to better understand what improves or hampers Athena's performance.

To construct this method-level call graph, we use *call-graph* [1], which is a tool that allows for generating a call graph of a software system and supports multiple programming languages.

Representation Fusing

Previous work suggest fusing semantic and structural information by taking a linear combination of the two coupling metrics.[10] However, this has its drawbacks as discussed previously.[17] For example, if one of the metrics performs poorly for a certain type of coupling, *i.e.,* structural coupling when evaluating methods that only have a hidden dependency, then it will drag down the overall performance of the fused metric. To overcome this drawback, we apply a content-aware GNN to combine the two sources of information in a non-linear manner. Specifically, we use the GraphSage[14] GNN that considered the features of the nodes of the graph for non-linearly combining both Neural-based and Structural representations to generate node vectors. This usage of content-aware GNNs has shown promise in the Biomedical domain for embedding Biomedical ontologies through work by Kotitsas *et al.*[18] Our novel contribution is through the use of a pretrained Transformer model for generating the semantic embedding of the nodes, the usage of GraphSage over node2vec[11] as done in Kotitsas *et al.*,[18] and application of this approach to the software engineering domain. We hypothesize that having this non-linear interaction, from two different information sources, allows GraphSage to learn hidden semantic and structural patterns at the method-level granularity.

Unlike node2vec, which learns node embedding

[1] https://github.com/WM-SEMERU/call_graph

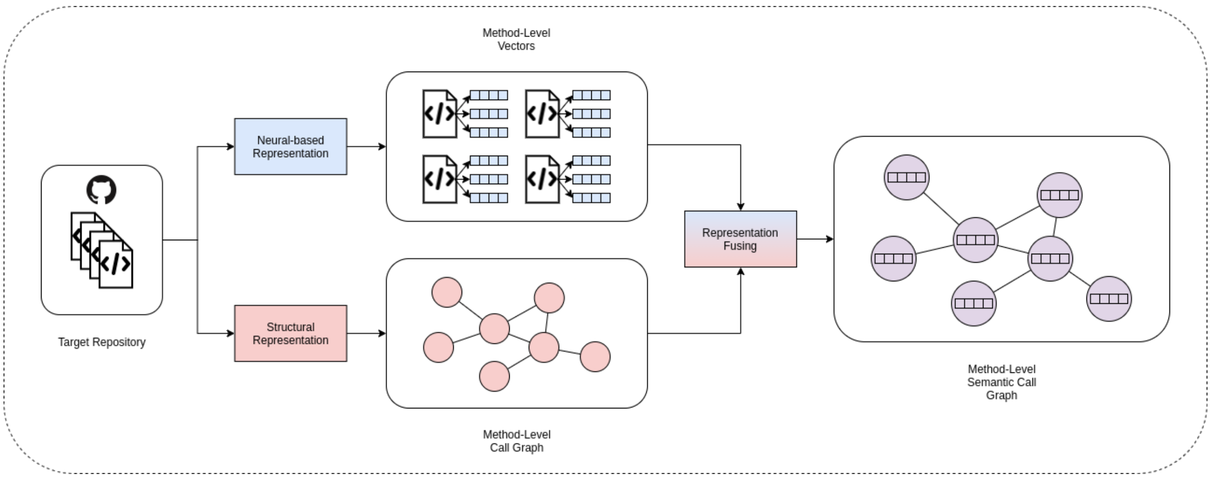
Method-Level Vectors
Neural-based Representation
Target Repository
Structural Representation
Representation Fusing
Method-Level Semantic Call Graph
Method-Level Call Graph

Figure 1: The Athena approach.

Algorithm 1: GraphSage Algorithm Overview from original paper.[14]

```
Input: Some Call Graph $G = (N, E)$ with node embeddings $x_n, \forall n \in N$
Output: Updated node embeddings $z_n, \forall n \in N$
1 $h_n^0 := x_n, \forall n \in N$
  /* loop through the number of aggregation functions */
2 for $k := 1...K$ do
      /* loop through each node in the graph */
3     for $n \in N$ do
          /* Collect all the node embeddings from the neighborhood of n */
4         $h_{\mathcal{N}(n)}^k := Aggregate_k(h_m^{k-1}, \forall m \in \mathcal{N}(n))$
          /* Concatenate the representation of the previous aggregation function and
             the current one and transform it by the weight matrix $W^k$ */
5         $h_n^k := (W^k \cdot CONCAT(h_n^{k-1}, h_{\mathcal{N}(n)}^k))$
      /* Normalize the vector to unit length */
6     $h_n^k := h_n^k / \|h_n^k\|_2$
  /* Return the final representations of all nodes */
7 $z_n := h_n^K, \forall n \in N$
```

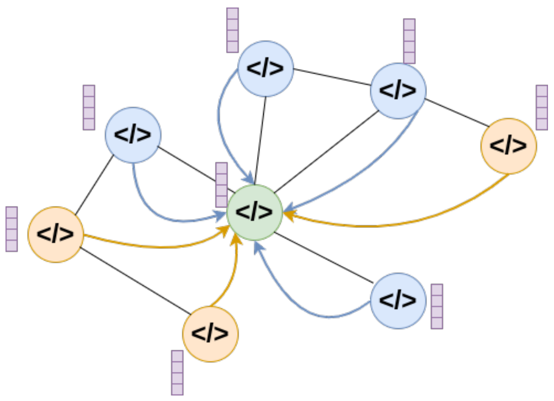

Figure 2: How GraphSAGE works.

that are specific to a single graph via a skip-gram paradigm to make node embeddings that are close to each other in the graph to have similar embeddings, GraphSage learns aggregation functions that takes an existing node embedding and transforms it based on its neighbors' node embeddings.

This process is outlined in Algorithm 1 with detailed comments for each step. This learning of aggregation functions, the W^k matrix in the above Algorithm, rather than the node embeddings directly allows GraphSage to be trained on a set of graphs and then tested on graphs it has never seen, which is not possible with node2vec. More formally, GraphSage optimizes the neural network, i.e., weight matrices W^k, using the following loss via stochastic gradient descent and backpropagation:

$$-log(\sigma(\mathbf{z}_n^T \mathbf{z}_m)) - \mathbf{Q} \cdot \mathbb{E}_{m_{neg} \sim P_{neg}(m)} log(\sigma(-\mathbf{z}_n^T \mathbf{z}_{m_{neg}})) \quad (1)$$

This loss is applied to all z_n's in the graph G and pushes the node embeddings of nearby nodes in the graph close to each other while at the same

Table 1: Hyperparameters used for training ATHENA.

Hyperparameter	Value
# Nodes per Aggregate Function	10
# of Layers	2
# of Node Embedding Dimensions	768
# of Hidden Channels	64
Optimizer	SGD + ADAM
Learning Rate	$1E-5$
Batch Size	256
Epochs	50

time pushing nodes that are far away, which are sampled using the $P_{neg}(m)$ distribution, far apart by tuning the weight matrices W^k. Fig. 2 gives a visual representation of this Algorithm where the different colored arrows represent the different aggregation functions being used to combine the node embeddings.

Implementation Details

You can find our selected hyperparameters in Tab. 1. We used a pretrained sentence-transformer model [2] that was trained on the task of matching a method with its docstring, which was from the CodeSearchNet dataset. For implementing the *Representation Fusing* GraphSAGE model, we used Pytorch Geometric[7] and modified their example implementation [3]. For managing data and training our model, we used Pytorch Lightning.[5]

All training was done on an Ubuntu 20.04 server with a single A100 NVIDIA GPU with 40GBs of VRAM, 128 CPU cores, and 1TB of RAM.

Evaluation

This section describes the procedure for evaluating how effective ATHENA is at the task of code search. Specifically, we wish to measure the ability of ATHENA to return highly relevant software methods to a natural language query. To accomplish this, we used an existing benchmark in the software engineering field called CodeSearchNet Challenge.[16] This benchmark covers a total of 99 natural language queries in English and has associated relevant and irrelevant methods that have been labeled by human annotators across six programming languages, namely, Go, Java, JavaScript, PHP, Python, and Ruby. Since CodeSearchNet Challenge only involves methods and ATHENA works at the software system level, we modified the challenge to allow ATHENA access to the entire software system rather than only single methods. We discuss this process in the following subsection. To guide our evaluation, we proposed the following research question:

RQ$_1$: *How well is* ATHENA *able to find relevant methods to a given natural language query?*

Data Collection

3.1.1 Project Selection

We focused only on the Python portion of the CodeSearchNet Challenge due to Python being one of the most popular programming languages and being supported by the tool we use to generate the call-graph. To convert the single method CodeSearchNet Challenge into the software system version that ATHENA requires, we went through each method in the benchmark and found its associated GitHub repository url, downloaded the repository, and attempted to generate the software system's call-graph. Any repositories that we could not generate the call-graph for were removed. This resulted in a total of 2,098 repositories to evaluate ATHENA on.

To train ATHENA, we used 250 software systems that we obtained from the Python portion of the CodeSearchNet Dataset used for training models. We followed the same process of converting the single method dataset to one capable of training ATHENA via software system call-graphs.

Relevant Method Retrieval

To answer **RQ$_1$**, we selected a total of 2,098 repositories from the CodeSearchNet Challenge benchmark's Python portion. Each repository has the main method that has a relevancy score to a given natural language query that ATHENA will rank again other methods in other repositories for the same query. Since, the natural language query is not part of the call-graph, it is not immediately able to be vectorized by ATHENA to perform the cosine similarity calculation that we use to rank the relevancy of each method in the benchmark to the query. To overcome this issue, we inserted the natural language query as though it was documentation into the graph by adding edges between a node representing the query and each method. We then vectorized the natural language query using the same sentence-transformer model used for the method and docstring vectors. ATHENA is finally applied to this newly augmented graph with the natural language query node and updates each node based on the learned aggregation functions. Once each node has been updated, the natural language query and method from the original

[2]https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base

[3]https://github.com/rusty1s/pytorch_geometric/blob/master/examples/graph_sage_unsup.py

benchmark node vectors are compared using cosine similarity to get their relevancy to each other. Once this process is done for each method from the original benchmark across the different repositories they belong to, the similarity scores are ranked. This is then repeated for each natural language query so that we have a list of queries and a ranked list of the most relevant methods to each query as computed by ATHENA. These rankings are then compared to the ground truth. Specifically, we used Normalized Discounted Cumulative Gain (nDCG) to calculate how close ATHENA's performance is to the ground truth. nDCG is the same metric used by the original CodeSearchNet Challenge benchmark. It is calculated using the following formula:

$$\text{DCG} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{2}$$

$$\text{IDCG} = \sum_{i=1}^{|REL_p|} \frac{rel_i}{log_2(i+1)} \tag{3}$$

$$\text{nDCG} = \frac{DCG}{IDCG} \tag{4}$$

Where DCG is the non-normalized discounted cumulative gain and is a metric representing how relevant the top most ranked retrieved documents are. The normalized version is this value divided by the ideal discounted cumulative gain (IDCG) where the position of the returned documents are sorted by their relevancy such that the top relevant document is in the first position. Usually, this relevancy score is binary, either relevant (1) or irrelevant (0). However, for CodeSearchNet Challenge, there is a 0 to 3 scale of relevancy with 3 being the most relevant. This does not impact the nDCG calculation.

Results

In this section we discuss the results from our evaluation and the limitations of ATHENA.

Relevancy Results Discussion

Tab. 2 show the results of ATHENA and previous approaches for the Python portion of the CodeSearchNet Challenge benchmark. The results are not perfectly comparable since some of the methods needed to be thrown out to work with ATHENA due to the repository they came from being unable to have a call-graph generated. However, there should not be a large change in the numbers due to the low amount of methods needing to be thrown out. As shown, ATHENA still lags behind the other approaches that rely only on the method information, instead of the software system information, with the only exception being our baseline TF-IDF model. We believe this to be due to the size of the ATHENA model being significantly smaller than the other models, especially cpt-code **M**, and trained with many more examples. Additionally, there has been previous research showing that graph neural networks suffer from an over-smoothing issue.[19] Specifically, Graph Convolutional Networks, of which GraphSage is a specific kind, have a difficulty when increasing layers, both in training stability as well as over-smoothing the features of the nodes. This means that nodes tend to look very similar to their neighbors. This is a desirable quality for classification problems, where you may wish to label all nodes in a neighborhood with the same label. However, for search, this present a problem with nodes not having enough differentiation and so irrelevant results might be returned.

Table 2: Statistics of the different libraries used in our evaluation.

Model	**nDCG**
TF-IDF	58.3
Athena	75.7
CodeBERT	84.0
SentenceTransformer	84.4
GraphCodeBERT	87.9
cpt-code **M**	**99.9**

Threats to Validity

Internal Validity. The biggest threat to internal validity comes from the usage of the CodeSearchNet Challenge benchmark for evaluating the ability of ATHENA to perform code search. This threat is limited by having the labels given to the relevance of methods to natural language queries being manually annotated by developers. Additionally, this benchmark has been used extensively in the software engineering field for observing progress.

The accuracy of call-graph generation is another threat to interal validity as we heavily rely on the on it. The tool we use for generating it has been tests for different types of software systems and their call-graphs and has been manually verified for some projects. Therefore, we are fairly confident the call-graph construction process.

External Validity. Due to the small number of natural language queries, 99, present in the CodeSearchNet Challenge benchmark, it is not possible to say how generalizable an approach that performs well on this benchmark will perform on various other natural language queries a developer might use. Future work will be needed to curate larger sets of natural language queries, preferably in other languages besides English, to help create a more robust benchmark. An additional external validity threat is the usage of Python only for Athena. Since we did not evaluate on other programming languages, it is unclear whether it would work for programming languages such as Java or C. However, the sentence-transformer model we used to generate the initial node embeddings was trained on multiple programming languages, so we believe there would be similar performance among other programming languages with Athena.

Related Work

Existing approaches have attempted to solve code search using classical keyword matching for computing syntactical similarity,[1,2,4,21–24,33] *e.g.,* Term-Frequency Inverse Document-Frequency (TFIDF).[30] However, natural language and software have little overlap in their vocabulary and their structure. Therefore, previous techniques such as keyword matching and TFIDF that rely on finding commonalities in the vocabulary of queries and methods resulted in poor performance.[9,13,35,36]

To overcome this limitation a lot of recent techniques compute semantic similarity such as using WordNet for finding synonyms of words as in Li *et al.*[20] Sachdev *et al.*[29] used word2vec,[25] a neural network that uses machine learning to represent words as continuous vectors, to generate word embeddings for natural language and software. Additionally, many recent approaches have used deep learning for code search[3,6,12,15,16] using a variety of different types of architectures and learning methods. The closest to ours is work by Feng *et al.*[6] since we are using their CodeBERT model as the foundation for Athena's *Neural Representation* component. However, similar to the other deep learning approaches for code search, CodeBERT does not consider the context methods reside in. We extend their work by including this additional context since software developers consider this important information when considering whether a method is relevant to a search query.[16] Lastly, recent work[26] has achieved remarkably high results on the CodeSearchNet Challenge showing that scaling up models and training data really helps the task of code search as well as showed we as a community may need to work on a new more challenging benchmark.

Conclusion and Future Work

In this paper we presented Athena, a Graph Neural Network that using a software project knowledge graph to learn the semantic and syntactic information present in the software project's methods. This learned information is then leveraged to retrieve methods relevant to a given query method. We evaluated Athena on a modified version of the popular CodeSearchNet Challenge benchmark using the common Normalized Discounted Cumulative Gain (nDCG) metric for search tasks and found Athena still lags behind previous state of the art methods. Additionally, we discussed potential reasons for why the worse performance and the underlying issue with Graph Convolutional Networks that Athena uses.

Future work. For future research, we intend to look into other model architectures besides Graph Neural Networks that can still leverage graph data such as GraphBERT.[34] GraphBERT uses a standard Transformer, but modified the input of graphs to work with the architecture. We hope this will overcome the over smooth issue that is present in Graph Neural Networks while still keep the additional context that a software system's call-graph gives the model. Additionally, we will be performing more in-depth analysis of the current Athena model and future Athena models to better understand success and failure cases.

Acknowledgements

I thank my other contributors David Nader, Daniel Rodriguez-Cardenas, Emad Aghajani, and Yanfu Yan for their help on this project with data collection and processing and on the evaluation. I also thank my advisor, Denys Poshyvanyk, for his help and guidance on this project. Finally, I thank the Virginia Space Grant Consortium for helping fund this research.

References

[1] Krugle code search https://www.krugle.com/, 2020.

[2] Ohloh code search https://code.ohloh.net/, 2020.

[3] Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. 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* (New York, NY, USA, 2019), ESEC/FSE

2019, Association for Computing Machinery, p. 964–974.

[4] Chan, W.-K., Cheng, H., and Lo, D. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2012), FSE '12, Association for Computing Machinery.

[5] Falcon, W., and .al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning 3* (2019).

[6] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online, Nov. 2020), Association for Computational Linguistics, pp. 1536–1547.

[7] Fey, M., and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[8] Firth, J. R., 11 1957.

[9] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The Vocabulary Problem in Human-System Communication: an Analysis and a Solution. *Communications of the Acm (cacm) 30*, 11 (1987), 964–971.

[10] Gethers, M., and Poshyvanyk, D. Using relational topic models to capture coupling among classes in object-oriented software systems. In *2010 IEEE International Conference on Software Maintenance* (2010), pp. 1–10.

[11] Grover, A., and Leskovec, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016).

[12] Gu, X., Zhang, H., and Kim, S. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, Association for Computing Machinery, p. 933–944.

[13] Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., and Menzies, T. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)* (May 2013), pp. 842–851. ISSN: 1558-1225.

[14] Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NeurIPS'17, Curran Associates Inc., p. 1025–1035.

[15] Husain, H., and Wu, H.-H. Towards natural language semantic code search https://githubengineering.com/towards-natural-language-semantic-code-search/, 2018.

[16] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436 [cs, stat]* (Sept. 2019). arXiv: 1909.09436.

[17] Kagdi, H., Gethers, M., Poshyvanyk, D., and Collard, M. L. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering* (2010), pp. 119–128.

[18] Kotitsas, S., Pappas, D., Androutsopoulos, I., McDonald, R., and Apidianaki, M. Embedding biomedical ontologies by jointly encoding network structure and textual node descriptors. In *Proceedings of the 18th BioNLP Workshop and Shared Task* (Florence, Italy, Aug. 2019), Association for Computational Linguistics, pp. 298–308.

[19] Li, Q., Han, Z., and Wu, X.-M. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence* (2018).

[20] Li, X., Wang, Z., Wang, Q., Yan, S., Xie, T., and Mei, H. Relationship-aware code search for javascript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 690–701.

[21] Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., and Baldi, P. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov. 18*, 2 (Apr. 2009), 300–336.

[22] Lv, F., Zhang, H., Lou, J., Wang, S., Zhang, D., and Zhao, J. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on*

Automated Software Engineering (ASE) (2015), pp. 260–270.

[23] McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., and Xie, Q. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering 38*, 5 (2012), 1069–1087.

[24] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, Association for Computing Machinery, p. 111–120.

[25] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[26] Neelakantan, A., Xu, T., Puri, R., Radford, A., Han, J. M., Tworek, J., Yuan, Q., Tezak, N., Kim, J. W., Hallacy, C., et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

[27] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning* (2021), PMLR, pp. 8748–8763.

[28] Reimers, N., and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing* (11 2019), Association for Computational Linguistics.

[29] Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., and Chandra, S. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (New York, NY, USA, 2018), MAPL 2018, Association for Computing Machinery, p. 31–41.

[30] Salton, G., and McGill, M. J. Introduction to modern information retrieval.

[31] Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[32] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, u., and Polosukhin, I. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NeurIPS2017'17, Curran Associates Inc., p. 6000–6010.

[33] Zhang, H., Jain, A., Khandelwal, G., Kaushik, C., Ge, S., and Hu, W. Bing developer assistant: Improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 956–961.

[34] Zhang, J., Zhang, H., Xia, C., and Sun, L. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140* (2020).

[35] Zhao, L., and Callan, J. Term necessity prediction. In *Proceedings of the 19th ACM international conference on Information and knowledge management - CIKM '10* (Toronto, ON, Canada, 2010), ACM Press, p. 259.

[36] Zhao, L., and Callan, J. Automatic term mismatch diagnosis for selective query expansion. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '12* (Portland, Oregon, USA, 2012), ACM Press, p. 515.