A Graph-Based Approach for Semantic Code Search

Horace Chu
Department of Computer Science
Stanford University
Stanford, CA 94305
horace@stanford.edu

Jerry Qu
Department of Computer Science
Stanford University
Stanford, CA 94305
jerryqu@stanford.edu

Andrew Tan
Department of Computer Science
Stanford University
Stanford, CA 94305
azt@stanford.edu

Abstract

Code search is a challenging problem that involves both natural language processing and information retrieval. In this paper, we propose a model that combines graph-based machine learning with transformer models and traditional search engines like Elasticsearch and apply it to CodeSearchNet, a competition for semantic code search. We apply hyperbolic embeddings and graph clustering on our data to incorporate the graph structure of code repositories into the search model. Our model is evaluated using normalized discounted cumulative gain (NDCG) and outperforms baseline language models and traditional search engines.

1 Introduction

Natural language processing has seen tremendous progress in the last few years. However, its application and success on source code is dwarfed by its success in more traditional language tasks. Software developers spend much of their time searching for and understanding how to use pre-existing code. Traditional search engines are not fit for indexing code, so we set out to use modern techniques in graphical machine learning and language models to improve code search.

Semantic code search is the task of retrieving relevant code snippets given a natural language query. While related to other information retrieval tasks, it requires bridging the gap between the language used in code (often abbreviated and highly technical) and natural language more suitable to describe vague concepts and ideas [HWG+19].

The CodeSearchNet Challenge (Husain et al., 2019) [HWG+19] is an attempt to explore and study semantic code search. Although the 21st century has witnessed the proliferation of numerous search engines that allow users to look up anything ranging from the price of the newest iPhone to the entire text of the Bible, there remains a marked absence of reliable and consistent code search capabilities. As the authors of this challenge noted, code search shares many similarities with other information retrieval tasks, but crucially requires bridging highly technical coding language with natural language understanding, which often consists of vague concepts and ideas.

2 Problem Definition

Since there is little shared vocabulary between search queries and code results, standard information retrieval algorithms often perform poorly in this task. Significant advances in deep learning and natural

language processing, combined with a drastic increase in computational power and the emergence of large datasets, have paved the way for new approaches in the study of this task. Correspondingly, we propose and analyze a method to increase the quality of semantic code search for users in all contexts.

The problem we are attempting to solve is the CodeSearchNet Challenge. Given a set of queries, we will build a model that returns (a set of) code functions with the intended functionality. We will evaluate the performance of our models using NDCG (normalized discounted cumulative gain), which allows for fairer comparisons between search engines for specific queries, and thus forms the basis for the evaluation of our models in this project.

3 Related Work

3.1 Approach

3.1.1 Embedding in Hyperbolic Space

Hyperbolic embeddings (De Sa et al., 2018) [DSGRS18] is a method of embedding graphs. It is particularly effective at embedding hierarchical structures, such as the trees we have in this dataset. The use of a hyperbolic embedding allows us to achieve low distortion and high precision embeddings using few dimensions.

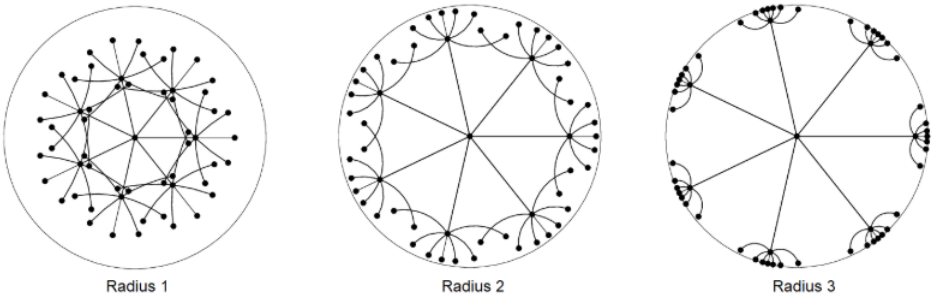

Figure 1: Examples for tree-like graphs embedded in hyperbolic space [DSGRS18]

3.1.2 Neural Bag-of-Words

Neural Bag-of-Words (NBOW) [SIFL16] is one of the most common architectures used in natural language processing tasks. Given some natural language snippet, we transform it into a dictionary where the keys represent unique words that are present in the natural language and the values represent the number of times it appears in the snippet. For example, if the sentence is "I am the son of the mayor," then the NBOW model would return 'I':1, 'am':1, 'the':2, 'son':1, 'of':1, 'mayor':1. In our baseline, we use this to approach convert each query token into a vector embedding that maps words to the frequency the word appears in the query.

3.2 Transformer Language Models

3.2.1 Attention

Attention Is All You Need (Vaswani et al., 2017) [VSP+17] introduces the Transformer, a seq-to-seq model trained solely on attention mechanisms instead of the traditional recurrent and convolutional methods. The Transformer architecture was found to be more parallelizable, require significantly less time to train, and generalize better to other language modeling tasks. The key architecture improvement is replacing the model's recurrent layers with multi-headed self-attention to derive global and local dependencies between input and output. The Transformer uses stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. We incorporate this approach in our baseline.

3.2.2 GPT-2

Language Models are Unsupervised Multitask Learners (Radford et al., 2019) [RWC+19] is a paper by OpenAI that introduces GPT-2, a 1.5B parameter Transformer model that achieves state-of-the-art results in language modeling tasks in a zero-shot setting. The model is trained with the simple objective of predicting the next word given all the previous words within some text. The simplicity of the objective allows it to be applied in a variety of settings, including synthetic text generation, complex reading comprehension, and summarization. Some of these tasks were even learned from the raw text, using no task-specific training data. The key takeaway is that natural language processing tasks, which were typically approached with supervised learning on task-specific datasets and architectures, could actually be approached without any explicit supervision and can be trained on never before seen datasets. GPT-2 has shown promise on non-language formats and has been successfully applied to source code with the Deep TabNine project [Jac19]. Projects in semantic code search have found success by leveraging the results shown in this paper.

4 Method

4.1 Dataset

The CodeSearchNet Challenge makes use of a dataset containing 2 million function-documentation pairs and 4 million other functions with no associated documentation, and is split 80-10-10 into the train, validation, and test sets, respectively. 99 commonly searched queries are chosen, and 10 candidate methods are generated for each query. For each query-function pair, an expert annotator evaluated how relevant the results are to the query and gave it a score from 0-3, with 3 being the most relevant. Code results span a number of programming languages including Python, Go, Ruby, Java, Javascript and PHP.

The training data is stored in jsonlines format where each line represents one example, a function with an associated comment. The data schema is as follows [HWG+19]:

- **repo:** the owner/repo
- **path:** the full path to the original file
- **func_name:** the function or method name
- **original_string:** the raw string before tokenization or parsing
- **language:** the programming language
- **code:** the part of the original_string that is code
- **code_tokens:** tokenized version of code
- **docstring:** the top-level comment or docstring, if it exists in the original string
- **docstring_tokens:** tokenized version of docstring
- **sha:** SHA hash code
- **partition:** a flag indicating what partition this datum belongs to of train, valid, test, etc. This is not used by the model. Instead we rely on directory structure to denote the partition of the data.
- **url:** the URL for the code snippet including the line numbers

Models are evaluated based on the 10 candidate functions proposed for each of the 99 test queries. In this paper, we focus on the Python dataset for evaluating our method, though the methods we propose can be applied to the other 5 programming languages.

4.2 Baseline

The baseline language model makes use of joint embeddings of code and queries. We used one encoder for every programming language, and trained the architecture to map the code and the corresponding language into a joint vector space. To identify the list of code snippets that are associated with a particular query, we first embed the query and then return the list of code snippets that have a close distance to it in the embedding space. The embedding functions used to achieve this are designed as follows:

1. The input sequences are preprocessed and code identifiers appearing in code tokens are separated out into sub-tokens
2. Natural language tokens are split into sub-tokens using byte-pair encoding
3. The resulting token sequences are processed using a number of different architectures to obtain token embeddings

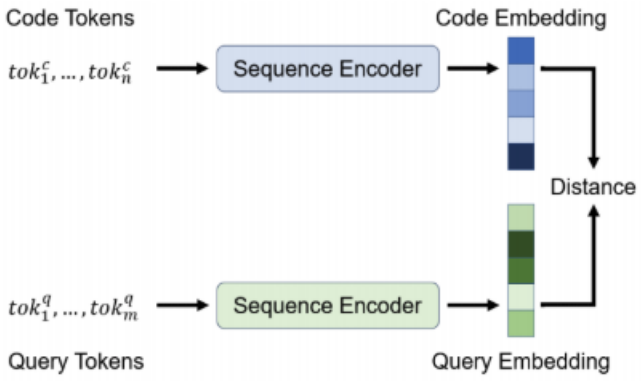

Figure 2: Architecture for generating the query and code embeddings [HWG+19]

For training, we have N pairs (c_i, d_i) of code and descriptions. We have a code encoder E_c and query encoder E_q. The algorithm is trained by minimizing the loss:

$$-\frac{1}{N}\sum_i log\frac{exp(E_c(c_i)^T E_q(d_i))}{\sum_j exp(E_c(c_j)^T E_q(d_i))}$$

During test time, we embed the 99 evaluation queries and code in the same space. For each query embedding, we find the 10 closest code embeddings as our ranked predictions.

4.3 Elasticsearch

For our approach, we use Elasticsearch as a basis. Elasticsearch is a distributed, full-text search and analytics engine. We use the Elasticsearch API to ingest our raw data, and perform full-text search to match queries with code snippets. We configure Elasticsearch with an index on the following fields:

1. **func_name:** the function or method name
2. **code:** the segment of the original_string that contains code
3. **docstring:** the top-level comment or docstring, if it exists in the original string

We use the standard Elasticsearch tokenizer and full-text search with fuzzy string matching to produce results and their corresponding scores. If the direct search of a query does not produce enough matching results, we relax the search by performing fuzzy string matching, which returns results that contain terms in the index that are similar to the query based on Levenshtein edit distance [YB07]. The default settings for edit distance is 0 for strings of one or two characters, 1 for strings of three to five characters, and 2 for strings of more than five characters. We find that an edit distance of 2 or more tends to return results that aren't relevant, so an edit distance of at most 1 is most common. If additional relaxation is required, we can further break down the query string into smaller query chunks. For each query, we generate over a hundred results. These results are ranked based on their scores, and the top 10 results for each query are used as our final predictions.

Here is an example of an Elasticsearch `match` query performed on our index `python_data_index`. We search for `"convert int to string"` on the `docstring` index with a `fuzziness` of 1.

```
GET python_data_index/_search?pretty
{
    "query": {
        "match": {
            "docstring": {
                "query": "convert int to string",
                "fuzziness": 1
            }
        }
    }
}
```

4.4 Capturing the Graph Structure

Our Elasticsearch-based approach currently does not utilize the fact that all functions are contained within a dataset of code repositories. To capture this information, we build a graph of hierarchy contained in the repositories. For each repository, we represent the directory hierarchy of files and functions as a tree. Non-leaf nodes represent folders or files and leaf nodes represent specific functions within a file. A directed edge from two nodes A to B means file/subfolder B sits directly within folder A (1 layer deep). The following were two approaches we considered:

1. Method 1: We create a tree for every unique repository. The root of the tree is the root of the repository, while each parent is a folder, and each child is a subfolder or file within the parent's folder. The leaves represent individual functions. This results in many disjoint trees, one for each repository in our dataset.
2. Method 2: We create a single connected tree for all the files. Using the disjoint trees from Method 1, we connect every single tree with a common root, creating a single connected tree.

We chose the latter as we wanted our embedding to be expressive enough to capture information across multiple repositories. Thus, we construct the graph by creating a tree for each repository and combine these trees into one single connected tree by joining their roots to a common root node.

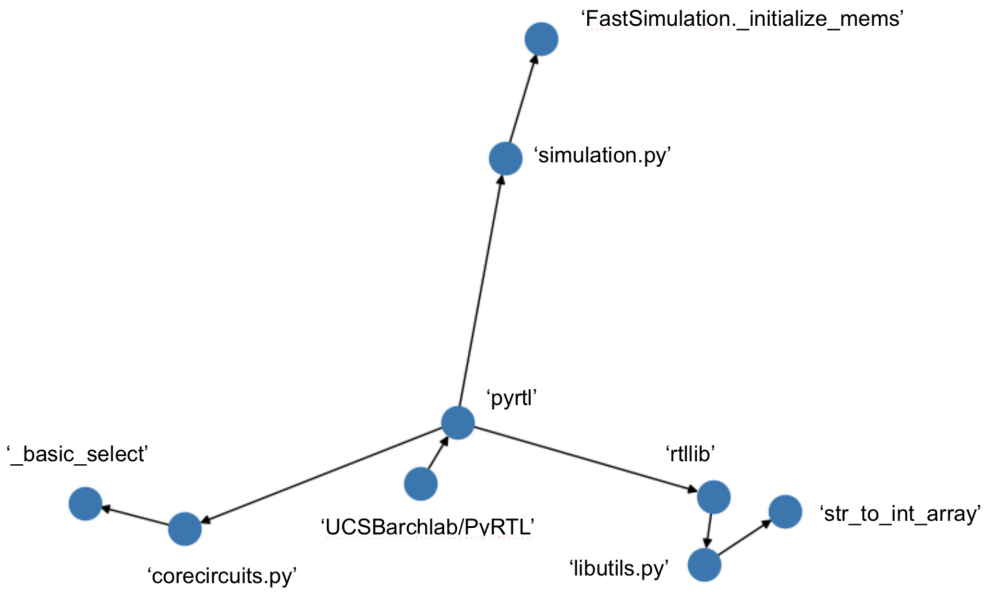

Figure 3: Example tree for a unique repository (Method 1)

4.4.1 Constructing Embeddings

To capture this graph in our model, we embed the nodes in our graph in a hyperbolic space. Our choice in the embedding allows us to capture with high precision the structure of our graph while using few dimensions. The low dimensionality is especially useful for learning.

This embedding preserves distances between nodes in our graph and is particularly effective at capturing hierarchical graphs such as trees. In our approach, we often have trees that have a high branching factor, as there may be many files in a directory, and many functions contained in a file. Further, our trees tend to be short, as file directories are rarely more than a few folders deep. For these short and bushy trees, a hyperbolic embedding is especially effective.

We construct embeddings using the combinatorial construction introduced by De Sa et. al. in their paper [DSGRS18]. We can thus compute the distance between pairs of nodes x, y in our embedding space as follows:

$$d_h(x, y) = acosh\left(1 + 2\frac{\|x - y\|^2}{(1 - \|x\|^2)(1 - \|y\|^2)}\right)$$

4.5 Clustering

Using the embeddings, we perform clustering based on the ranked top 100 functions returned by Elasticsearch. We explore various clustering methods and distance metrics. For simplicity, we use the standard k-means algorithm with Euclidean distance. We also consider hyperbolic k-means with hyperboloid clusters. We choose 10 as our number of clusters for the 10 ranked results in our final prediction. The number of clusters is a hyperparameter that can be tuned based on the intention of the user. Fewer clusters implicitly suggests less diversity in the sampled predictions.

4.6 Elasticsearch with Hyperbolic Embedding-Based Clustering

Using the Elasticsearch model, we can generate over 100 scored and ranked code snippets for each query. However, these results are purely based on string matching and do not leverage the inherent tree structure of the package hierarchy in code. Also, results from text-based methods alone lack the diversity that strong search engines provides. Similar to how the top Google search results usually come from different websites and domains, we want our candidate set to exhibit diversity and surface results that methods like Elasticsearch alone wouldn't be able to uncover.

In our final model, we leverage the graph structure, embeddings, and clustering from the sections discussed above.

Algorithm 1: Cluster and Re-rank

Result: Top 10 functions for a query based on Hyperbolic Embeddings

```
def ReRank(data, query, num_clusters=10):
    data ← ElasticSearch(data) results
    kmeans ← KMeansCluster(data, num_clusters)
    group data by kmeans_label
    sorted_labels ← sort labels by average score DESC
    new_predictions ← []
    for label ∈ sorted_labels do
        data_label ← subset data on label
        sort data_label by score DESC
        top ← top result from data_label
        new_predictions.append(top)
    end
    return new_predictions
```

Algorithm 2: Cluster and Re-rank with Importance Sampling

Result: Top 10 functions for a query based on Hyperbolic Embeddings with Importance Sampling

```
def ReRankWithImportanceSampling(data, query, num_clusters=10):
    data ← ElasticSearch(data) results
    kmeans ← KMeansCluster(data, num_clusters)
    group data by kmeans_label
    sorted_labels ← labels and count sorted by average score
    new_predictions ← []
    N ← size of data
    for label, count ∈ sorted_labels do
        data_label ← subset data on label
        sort data_label by score DESC
        M ← number proportional to count / N (importance sample)
        top ← top M results from data_label
        new_predictions.append(top)
    end
    return new_predictions
```

5 Results and Discussion

5.1 Baseline

The NBOW baseline model achieved a NDCG score of 0.189848815. This is produced from the tuned baseline models from by the CodeSearchNet Challenge [HWG+19]. These results are based purely on the tokens generated from the code and docstring, and do not leverage any of the graph-based methods discussed.

GitHubRepo	Author	NDCG Average
GitHub Link	azt	0.189848815

5.2 Elasticsearch

The Elasticsearch model achieved a NDCG score of 0.195542301, which represents a 0.0057 improvement from the baseline. This suggests Elasticsearch by itself is insufficient to produce a substantial increase in performance.

GitHubRepo	Author	NDCG Average
GitHub Link	azt	0.195542301

To better understand the results we have from Elasticsearch, we have tried a couple of things. First, we are interested in the impact of ordering in our predictions. We took the top 10 results for each query produced by Elasticsearch and randomized the ranking. The randomized ranking achieved a NDCG score of 0.155259178, well below the score from the original ranking and below any of the baseline scores provided.

5.3 Elasticsearch with Graph-Based Clustering

Next, we explore the impact of our graph-based approach on performance. We hypothesized that by including a diverse set of functions, we might be able to produce higher quality results to a human. In this approach, we chose the top-scoring node from each cluster and included it in the ranking. The

average score of the cluster was used in ranking the functions, so that the top-scoring function from the cluster with the highest average score was ranked first.

Unfortunately, our NDCG score was 0.141029402, which is well below both the baseline and the model using only Elasticsearch. We can thus conclude that in attempting to increase the diversity of our results, we may have inadvertently increased diversity too much and potentially introduced some functions that may not have been highly relevant to the query.

Our clustering will tend to cluster functions within one repository in the same cluster. In the case where the top results ought to all come from the same cluster, we are unnecessarily penalizing some nodes. For example, if we wanted to find code for traversing a graph, the most relevant functions may all be contained in a single file that contains all the functions for traversing a graph in a particular library. More generally, a single file may contain several related functions, a number of which may be highly relevant to the query. In this clustering however, we will be discarding all but the highest scoring of these functions.

Thus, our next approach with clustering is to perform importance sampling. We sample the top nodes proportional to the number of nodes in the cluster. For example, if a cluster contained half of the nodes in the graph, we sample half of the functions from this cluster. This allows for us to include a high amount of potentially highly-relevant functions from one file or repository, but very different results as well, which may be particularly useful when the query author's intent is uncertain.

Using this approach, we achieved a NDCG score of 0.208053383. This result is 0.090 better than the baseline and 0.0125 better than using Elasticsearch alone.

GitHubRepo	Author	NDCG Average
GitHub Link	azt	0.208053383

We can conclude that the re-ranking from the graph clustering provided a small improvement in performance. However, because of the increased diversity in rankings, we believe the results may be more useful and relevant for an actual human reader.

The Elasticsearch model returns many syntactically similar results because the search engine is purely based on the tokenized texts. For example, for the query "convert int to string," the Elasticsearch model returns many similarly worded functions in the top 10: `to_int`, `_int`, `string_to_int`, `to_str`, `_Meta.convert_to_integer`.

The graph-based model with clustering and importance sampling returns more diverse results for the same query: `int_to_string`, `int_to_decimal_str`, `to_str`, `SourceSpreadsheet._convert_value`, `convert_ifnum`.

We also note that the Elasticsearch model knows we are interested in converting between integers and strings, but not the direction of the conversion, as can be seen from the results it returns (e.g. `string_to_int` vs. `to_str`). We also get more unique results from the graph-based model such as `SourceSpreadsheet._convert_value` and `convert_ifnum`.

6 Conclusion

Overall, Elasticsearch with graph-based clustering provides an incremental improvement on the Elasticsearch-alone and baseline language models. Depending on the embedding, clustering, and re-ranking methods used, Elasticsearch with graph-based clustering can provide very different results. We hypothesize this can largely be attributed to the varying amounts of diversity inherent in the different configurations of our model. While wide diversity in predictions may not be the most optimal for maximizing NDCG scores, we believe diversity is code suggestions can be quite beneficial and more useful to an actual user of code search.

7 Future Work

Although using Elasticsearch and Hyperbolic Embeddings together gave us superior results to the baseline models or Elasticsearch on its own, there are many other potential improvements and modifications that could have been implemented if time allowed. In this section, we detail a few of these ideas.

- One of the strengths of Elasticsearch is its ability to represent rare words, which are very frequently found in code. Neural models that are able to efficiently render rare words are likely to perform better.
- In this project, we returned code snippets that are appropriate given the intent of the query, but we did not test for code quality. Doing so could reduce the possibility that bad results are returned, thus leading to higher quality code search results.
- Elasticsearch models tend to focus heavily on code identifiers, such as the function name or the variables present in the code snippet, but control flow could be an important and meaningful signal too. Future methods that leverage this information could lead to better results.
- The 99 queries we used in this project can be described as general purpose, since they are not specific to particular projects/purposes. Modifying search algorithms to account for more specialized queries could lead to a code search model that can be applied in different contexts, whether specific or general.
- We can explore alternative ways to use the graph structure in re-ranking the results, such as using spectral clustering.

Acknowledgments

We are immensely thankful to Michele Catasta for introducing us to the topic of code search using graphs and language models, as well as for his continued mentorship and guidance throughout this project. We would also like to thank GitHub for opening this challenge and making the data and tools available, in an effort to advance the field of semantic code search. Finally, we want to express our profound gratitude to Prof. Jure Leskovec and the entire CS 224W staff for the excellent class.

Team Contributions

Horace, Jerry, and Andrew contributed equally to this project.

References

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[DSGRS18] Christopher De Sa, Albert Gu, Christopher Ré, and Frederic Sala. Representation tradeoffs for hyperbolic embeddings. *Proceedings of machine learning research*, 80, 04 2018.

[HWG+19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. 09 2019.

[Jac19] Jacob Jackson. Autocompletion with deep learning, 07 2019.

[NJW02] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Adv. Neural Inf. Process. Syst*, 14, 04 2002.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[SIFL16] Imran Sheikh, Irina Illina, Dominique Fohr, and Georges Linarès. Learning word importance with the neural bag-of-words model. pages 222–229, 01 2016.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 06 2017.

[YB07] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1091–1095, June 2007.