Learning to search for software artifacts

Breandan Considine, Xujie Si, Jin Guo

April 21, 2021

1 Introduction

Humans are adept information foraging agents. We can quickly find relevant information in a large corpus by recognizing and following textual landmarks. Software projects are composed of a variety of semi-structured documents containing many clues where relevant information may be found. In this work, we train a model to navigate and read software artifacts like source code and documentation, in order to facilitate common programming tasks such as code search, completion, or defect prediction.

Early work in program learning realized the importance of graph-based representations [1], however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal [4], graphical [14] or other user interface to explore the space of valid programs, however do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph [9] from source code, but still require an explicit parser to form the initial graph and adaption in settings where style, terminology and document structure vary remains a challenge.

Imagine a newly-hired developer, who has programming experience, but no prior knowledge about a closed-source project. She receives access to the team's Git repository and is assigned her first ticket: Fix test broken by `0fb98be`. After locating the commit and becoming familiar with the code, she queries StackOverflow, discovers a relevant solution, copies the code into the project, makes a few edits, presses run, and the test passes.

In order to accomplish a similar task, an information-seeking agent must be able to explore a local database for information, construct a query, and search a remote database. Similar to a human developer, it might traverse the project to gather information, taking note of various keywords, APIs and design patterns. While traversing the project, the model constructs

a project-specific knowledge graph, which it uses to locate other relevant artifacts written by authors who solved the same problem.

2 Prior literature

Prior work in the code search literature explores the text-to-code [8] setting, where queries are typically considered to be a short sequence composed by the user, or code-to-code [10] setting where the query might be generated. Model performance typically evaluated using mean reciprocal rank (MRR), mean average precision (MAP), normalized discounted cumulative gain (NDCG), precision and recall at top N-results (P/R@N), or similar metrics. Although some [2] do incorporate other features from the local document, none however, consider the query in the context of a broader project. The ability to align contextual features from the surrounding project, we argue, is essential to delivering semantically relevant search results.

Name	Type	Key contribution	Evaluation method
Our Work	Project-to-code	Graph search	TBD
FaCoY [10]	Code-to-code	Similarity search	MRR
Yogo [13]	Code-to-code	Equational reasoning	Case study
CSNet [8]	Text-to-code	Big code benchmark	MRR
Aroma [11]	Code-to-code	Structural Search	Recall@N
AUSearch [2]	Code-to-code	Type resolver	Highlight accuracy
Cambronero [3]	Text-to-code	Semantics	Answered@N
DeepCS [6]	Text-to-code	Deep learning	FRank, *@N, MRR

Table 1: Comparison of existing code search tools.

Although we draw some inspiration from the duplicate detection literature, their work presumes the code was intentionally duplicated or modified but is essentially identical in form or function. In contrast, our work seeks to help users discover artifacts which predictive of as-yet-incomplete code. By aligning salient features from the surrounding document and project graph, we approximate the notion of semantic similarity, without requring a language-specific parser like some methods [3] do, and can more easily adapt to settings where the language and project structure varies.

Furthermore, unlike prior work, our work considers the surrounding project and related artifacts. Instead of parsing their contents explicitly, which may be computationally intractable, we allow the agent to construct the graph organically by exploring the filesystem, then extract a query.

3 Method

We fetch a dataset of repositories sampled repositories on GitHub, containing a mixture of filetypes representing source code and natural language artifacts. From each repository, we index all substrings of every line in every file using a variable height radix tree producing a multimap of `kwdIndex: String -> List<Location<F, O>>` of `String` queries to file-offset pairs. We also encode CodeBERT [5] sequence embeddings to substrings `knnIndex: Vector -> Location<F, O>` using a Hierarchial Navigavble Small World Graph [12] (HNSWG).

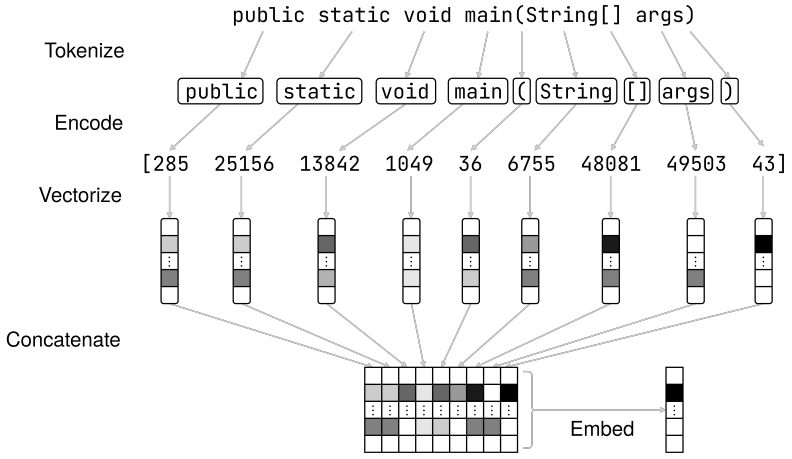

Figure 1: CodeBERT takes a unicode sequence and emits a vector sequence, which we accumulate into a single vector.

For each token in a string, CodeBERT emits a length-768 vector, so a line with n tokens produces a matrix of shape $\mathbb{R}^{768\times(n+1)}$, the first column of which contains the final hidden layer activations. We concatenate the CodeBERT sequence '`<s>`' and using the source tokens, encode and vectorize the encoded sequence using the vocabulary, then take the first row as our sequence embedding as depicted in Fig. 1. In § 6, we compare various distance-metrics to fetch the nearest sequence embeddings in our database

and compare precision and recall across various types of distance metrics.

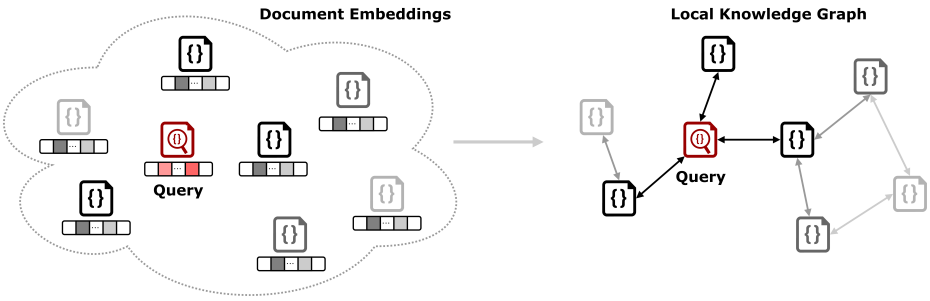

Figure 2: To compute our query neighborhood, we traverse the HNSWG up to max depth d, i.e. $d = 1$ fetches the neighbors, $d = 2$ fetches the neighbors-of-neighbors, and $d = 3$ fetches the neighbors-of-neighbors-of-neighbors.

For a given query and context, we first compute the context embedding and using a distance metric, fetch the k-nearest neighboring documents in latent space, forming the depth-1 nodes in our graph, then repeat this procedure recursively, pruning with a beam search heuristic based on the total path length in latent space. This procedure is depicted in Fig. 2.

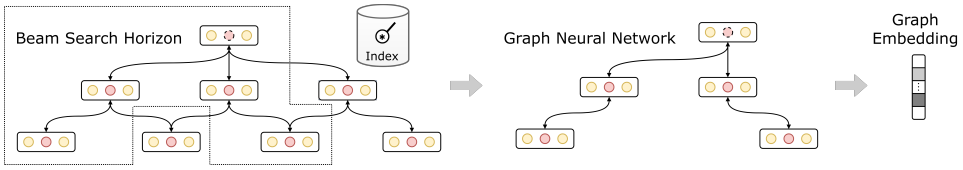

Figure 3: Unlike language models which directly learn the data distribution, our model is designed to query an unseen database only available at test time. The model scores and selectively expands a subset of promising results within the database using a beam search heuristic, then runs message passing over the resulting graph to obtain the final task prediction.

Once the beam search procedure is complete, we have a graph representing the neighborhood of the nearest neighboring code fragments in the parent project. This forms a so-called *virtual knowledge graph*, with nodes decorated by the CodeBERT sequence embeddings and edges decorated with

the direction vector between documents. We then run p rounds of message passing to obtain graph embedding or *neighborhood summary vector* (Fig. 3).

4 Experiments

In this work, we attempt to understand the relationship between entities in a software project. Our research seeks to answer the following questions:

1. Which contexts in a software project share mutual information?
2. To what degree can we claim the model has learned to:
   (a) Locate contextually relevant artifacts within a software project?
   (b) Comprehend the semantic content of the artifacts traversed?
   (c) Apply the knowledge gathered to perform the assigned task?

In contrast with classical code completion models which only require a file-local context, our method is designed to navigate an entire project. In the following experiments, we compare completion accuracy with a vanilla sequence prediction model, as well as an AST-structured sequence prediction model trained from a corpus of Java projects on the same task.

We hypothesize that by jointly learning to choose locations in a project over which to attend while solving a downstream task, such as masked sequence completion, our model will produce a feature representation capable of locating and extracting information from semantically relevant contexts. We evaluate our hypothesis both qualitatively and quantitatively.

In our first set of experiments, we try to understand what is shared between sequences mapped to similar latent vectors. Do similar sequences share salient keywords? Are those keywords relevant to the task?

In our second experiment, we try to measure the information gain from by including and excluding various filetypes through ablation. For graphs containing filetypes which include Markdown or Java, what kinds of information do these resources provide and which categories are most salient?

In our third and final set of experiments, we compare performance across hyperparameters. Does contextual expansion lead to better task performence for a given sequence prediction task? By relaxing edge-construction criteria and increasing hyperparamers such as beam search budget, we would expect corresponding task performance to increase.

If our hypothesis is correct, the virtual knowledge graph will span both natural language and source code artifacts. If so, this would provide evidence

to support the broader hypothesis [7] that documentation is a useful source of information. In addition to being useful for the prediction task itself, we anticipate our model could also be used for knowledge graph extraction and suggest semantically relevant code snippets to developers.

5 Evaluation

We evaluate on the following loss function, adapted from [8]. Given a set of code and context pairs, $\mathbf{F} = (\mathbf{c}_i, \mathbf{d}_i)_{i=1}^{N}$, where $\mathbf{S} \sim_{i.i.d.} \mathbf{R}_d$:

$$\mathcal{L} = \frac{1}{N} \sum_{i}^{N} \log \left(\frac{\langle \varphi(\mathbf{c}_i), \varphi(\mathbf{d}_i) \rangle}{\sum_{j \neq i}^{N} \langle \varphi(\mathbf{c}_j), \varphi(\mathbf{d}_i) \rangle} \right) \tag{1}$$

In other words, we minimize the distance between true code and context pairs, while maximizing the distance between distractors $(\mathbf{c}_j, \mathbf{d}_i)_{i \neq j}$.

6 Results

Our preliminary results compare distance metrics (Fig. 4), explore embedding quality (Fig. 5) and visualize the synthetic knowledge graphs (Fig. 6).

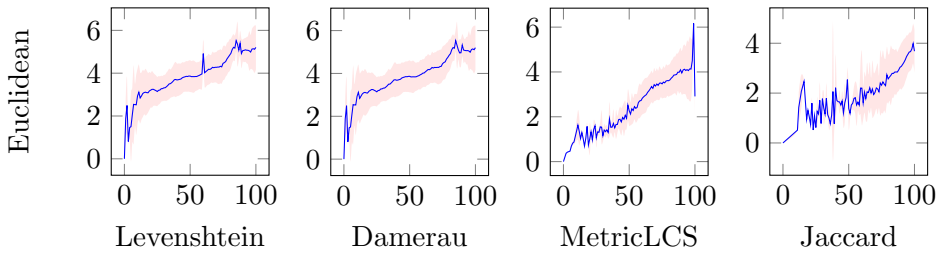

Figure 4: CodeBERT latent space distance correlates with string distance.

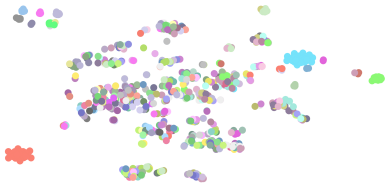

Figure 5: Reduced dimensionality TNSE embeddings colored by line length.

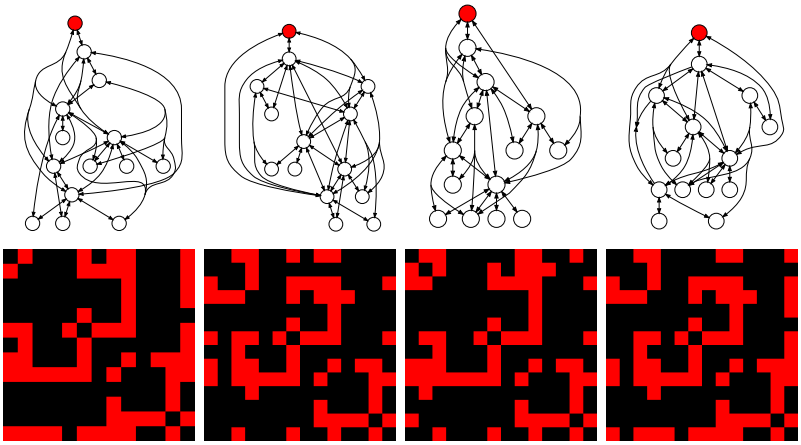

Figure 6: Example graphs sampled by beam search with $d = 10, w = 5$.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang. AUsearch: Accurate API usage search in GitHub repositories with type resolution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 637–641. IEEE, 2020.

[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*, pages 964–974, 2019.

[4] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. *arXiv preprint arXiv:1906.04604*, 2019.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[6] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[7] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14. IEEE, 2017.

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

[9] Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

[10] 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*, pages 946–957, 2018.

[11] 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):1–28, 2019.

[12] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[13] 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*, pages 1066–1082, 2020.

[14] Homer Walke, R Kenny Jones, and Daniel Ritchie. Learning to infer shape programs using latent execution self training. *arXiv preprint arXiv:2011.13045*, 2020.