Learning to navigate, read and apply software documentation like a human

Breandan Considine, Xujie Si, Jin Guo

March 30, 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 such clues where relevant information may be found. In this work, we train an agent to navigate and read software artifacts like source code and documentation, in order to facilitate common programming tasks such as code 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 [2], graphical [7] 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. Adapting to settings where style, terminology and document structure vary remains a challenge. Early work has shown the feasibility of learning a graph [5] from source code, but only locally and still requires an explicit parser to form the initial graph.

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 database. Similar to a human developer, it might be able to perform simple actions, like `search(query)`, `read(text)`, `copy(text, from, to)`, and `runCode()` while navigating relevant documents and source code artifacts. As the agent traverses the project using these primitives, it builds up a project-specific knowledge graph.

2 Method

We fetch a dataset of repositories sampled from popular Java 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 [3] sequence embeddings to substrings `knnIndex: Vector -> Location<F, O>` using a Hierarchial Navigavble Small World Graph [6] (HNSWG).

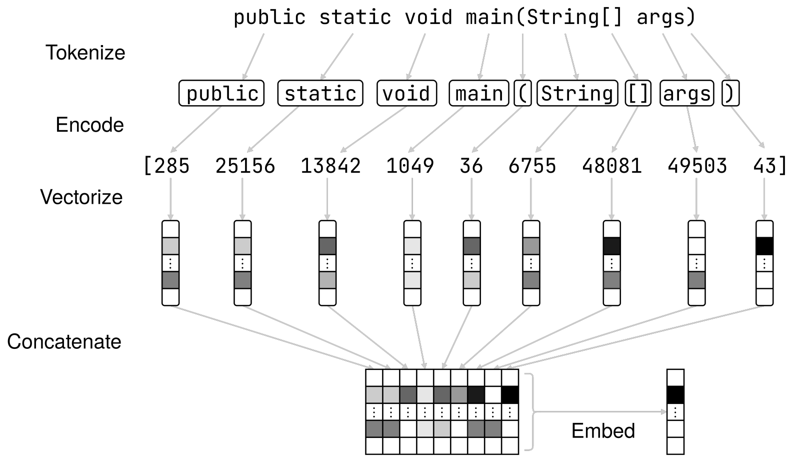

Figure 1: CodeBERT takes a unicode sequence and emits a vector sequence, which we accumulate into a single vector.

For each token in a string, CodeBERT emits a length-512 vector, so a line with n tokens produces a matrix of shape $\mathbb{R}^{512\times n}$, which we flatten into a vector. Once we have the sequence-embedding, we compare various distance-metrics to fetch the nearest sequence embeddings in our database. We then compare precision and recall across various types of distance metrics.

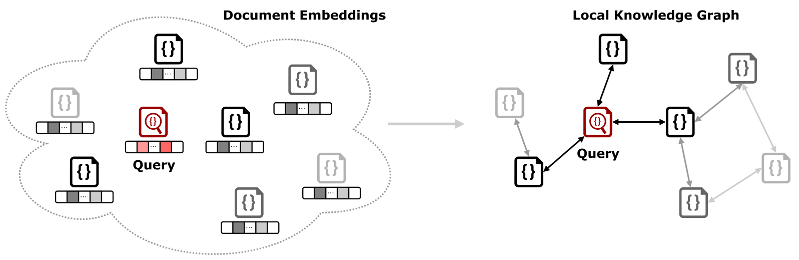

Figure 2: To compute the k-nearest neighbors for a given query, we traverse the HNSWG up to depth d, i.e. $d = 2$ retrieves the neighbors-of-neighbors, and $d = 3$ fetches the neighbors-of-neighbors-of-neighbors.

For a given query location, we compute the context embedding and fetch the k-nearest neighboring documents in latent space. The edges are decorated with the direction vector between neighbors. We then repeat the procedure recursively, pruning with a beam search heuristic based on the path length the latent space.

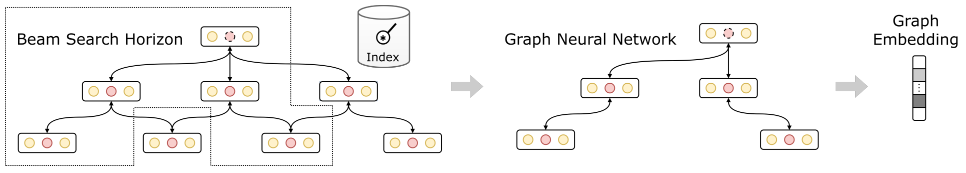

Figure 3: Unlike language models which directly learn the data distribution, our model is designed to query an unseen database only available at test time. The model scores and selectively expands a subset of promising results within the database using a beam search heuristic, then runs message passing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

Once the beam search procedure is complete, we have a graph of the nearest neighboring documents. This forms a virtual knowledge graph, with nodes decorated by the CodeBERT embeddings and edges decorate with the direction vector between the documents. We then run p rounds of message passing to generate a neighborhood summary.

3 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 agent 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 locations. We evaluate our hypothesis both qualitatively and quantitatively.

In our first experiment, we attempt to understand which queries the agent is performing while solving a programming task. Does it search for certain keywords in the context? Are those keywords relevant to the task?

In our second experiment, we try to understand the approximate relation between project artifacts and measure the information gain from various filetypes through ablation. For trajectories containing filetypes which include Markdown or Java, what kinds of information do these resources provide and which categories are most salient for the assigned prediction task?

In our third experiment, we compare prediction accuracy across architectures and datasets. Should we constrain the action space (e.g. only querying tokens from the surrounding context) for more efficient trajectory sampling, or allow arbitrary queries? How well does the model architecture transfer to new repositories, within and across programming languages?

If our hypothesis is correct, the policy network will learn to use both natural language and source code artifacts. If so, this would provide evidence to support the broader hypothesis [4] 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.

4 Next steps

Our next steps are to build a simple RL environment which allows an agent to interact with a software repository and construct a graph. We will use an in-memory filesystem to store and load project artifacts. The policy network will need to be pretrained on a corpus of projects in the same language. To model the action space, we will use the radix tree of the parent project, with transition probabilities conditioned on the local context.

For further details and to get started, please visit our GitHub page: `https://github.com/breandan/gym-fs`.

5 Results

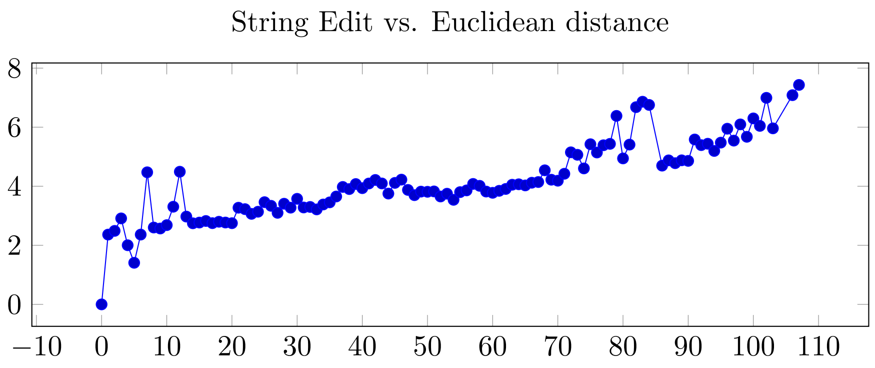

Figure 4: Levenstein edit distance and average Euclidean distance in CodeBERT latent space appears to be positively correlated.

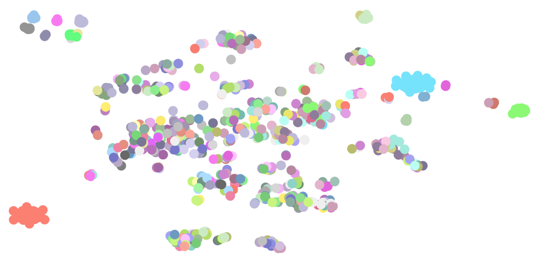

Figure 5: Reduced dimensionality TNSE embeddings colored by line length.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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