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

Breandan Considine, Xujie Si, Jin Guo

March 8, 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.

Our work broadly falls under the umbrella of text-based reinforcement learning. Prior literature falls under two categories: natural or formal language. Reinforcement learning (RL) in the natural domain typically focuses on question answering [4, 6], or interactive text games [10, 3, 12, 9, 2]. RL techniques have begun to show promising results for program synthesis [7, 11, 5]. Our work falls at the intersection of these two domains.

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 [7], graphical [13] 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 and adapting to settings where style, terminology and document structure vary remains a challenge. Early work has shown the feasibility of learning a graph [11] from source code, but only locally and still requires an explicit parser to form the initial graph.

Unlike prior work, we consider the whole project and related artifacts. Instead of parsing their contents explicitly, which may be computationally expensive or too large to fit in memory, we allow the agent to construct the

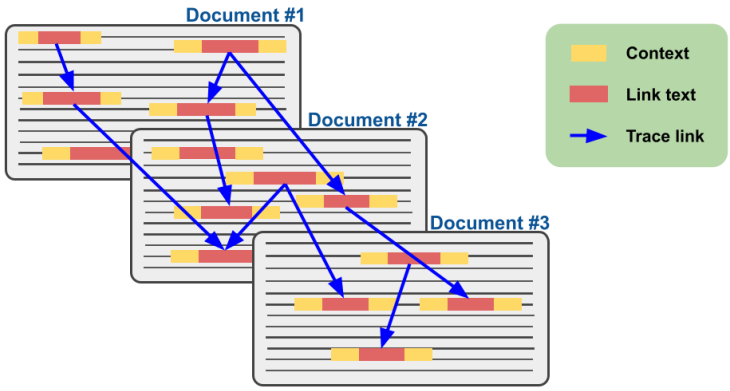

Figure 1: Software projects consist of many documents related by common keywords. Matching locations form an entity alignment graph, with vertices decorated by the surrounding context, and edges representing the relation type, e.g. an exact match, synthesized query or explicit parse result.

graph organically by exploring the filesystem, which can later be used to predict a label or sequence at inference time, depending on the task.

Imagine a newly-hired developer, who has programming experience, but no prior knowledge about a closed-source project. After onboarding, she gets access to the team's Git repository and is assigned her first ticket: Fix test broken by `0fb98be`. After locating the commit and getting familiar 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 search for resources in 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()` to navigate relevant documents and source code artifacts. As the agent traverses the project using these primitives, it builds up a project-specific knowledge graph.

During evaluation, we measure how well the agent performed. Many loss functions are possible depending on the task in question, from a simple string distance metric over a masked code fragment, to a more complex property (e.g. the presence or absence of an error, or some internal state of the REPL) which must be satisfied by interacting with the runtime environment. Many program analysis and repair tasks are amenable to the setup described, including defect, duplicate or vulnerability detection and correction.

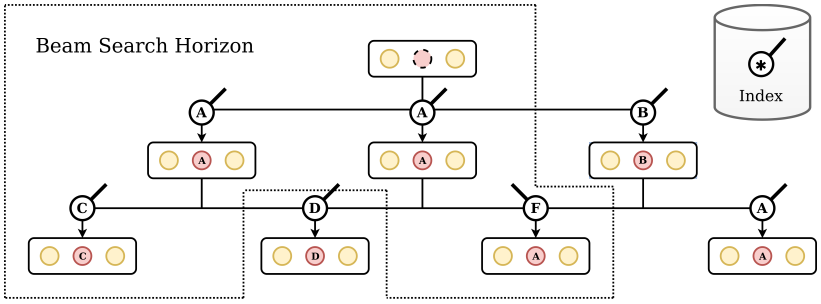

Figure 2: Unlike language models which directly learn the data distribution, our model is designed to query an external database only available at test time. The model scores and selectively expands a subset of promising results with the database using a beam search heuristic, then runs message passing over the resulting graph to obtain the final task prediction.

2 Method

We fetch a dataset of random repositories on GitHub containing a mixture of filetypes representing source code and natural language artifacts. Each project is then indexed using a variable height radix tree, producing a multimap of queries to file offset (F, O) pairs. Querying the index produces a list of *concordances* whose text matches the query.

We initialize the policy network using a pretrained language model. Starting at the site of the prediction task and conditioning on its local context, the policy network draws K queries from its latent state and queries the index to retrieve a set of salient locations within the parent project. This same process is repeated at each location using either direct BFS traversal, finite-horizon MCTS or beam search to retrieve a set of contextually relevant locations within the parent project.

The rollout traces form a graph of related locations inside the project and their corresponding context embeddings, which form the GNN node features. Once the rollout ends, we run message passing on the resulting GNN for a fixed numbers of steps, and then decode the graph embedding to produce the task prediction. Both the decoder, GNN weights, and policy network are trained end-to-end on the downstream task, e.g. code completion, defect detection or correction.

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 kinds of artifacts in a software project are visited most often?
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 our first experiment, we attempt to understand which queries the agent is performing when 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 measure the information gain from various filetypes through ablation. For trajectories containing filetypes such as Markdown or Java, what information gain do these resources provide and which filetypes are are the most salient for the prediction task?

In our third experiment, we compare prediction accuracy a variety of hyperparameters, such as search horizon depth, action space, and parameter sharing across query, GNN and decoder networks. Should we constrain the action space (e.g. by only considering query tokens from the surrounding context) for more efficient trajectory sampling, or allow arbitrary queries?

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 [8], that documentation is a useful source of information. In addition to being useful for the prediction task itself, this could also provide a mechanism 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`.

References

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

[2] Prithviraj Ammanabrolu and Matthew Hausknecht. Graph constrained reinforcement learning for natural language action spaces. *arXiv preprint arXiv:2001.08837*, 2020.

[3] Prithviraj Ammanabrolu and Mark O Riedl. Playing text-adventure games with graph-based deep reinforcement learning. *arXiv preprint arXiv:1812.01628*, 2018.

[4] Christian Buck, Jannis Bulian, Massimiliano Ciaramita, Wojciech Gajewski, Andrea Gesmundo, Neil Houlsby, and Wei Wang. Ask the right questions: Active question reformulation with reinforcement learning. *arXiv preprint arXiv:1705.07830*, 2017.

[5] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.

[6] Yu Chen, Lingfei Wu, and Mohammed J Zaki. Reinforcement learning based graph-to-sequence model for natural question generation. *arXiv preprint arXiv:1908.04942*, 2019.

[7] 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.

[8] 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.

[9] Xiaoxiao Guo, Mo Yu, Yupeng Gao, Chuang Gan, Murray Campbell, and Shiyu Chang. Interactive fiction game playing as multi-paragraph reading comprehension with reinforcement learning. *arXiv preprint arXiv:2010.02386*, 2020.

[10] Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep reinforcement learning with a natural language action space. *arXiv preprint arXiv:1511.04636*, 2015.

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

[12] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*, 2015.

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