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

Breandan Considine, Xujie Si, Jin Guo

February 24, 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 feature a variety of semi-structured documents containing many such clues indicating where contextually relevant information can be found. In this work, we train an agent to solve a programming task, by learning to navigate and read software artifacts like source code and documentation in order to facilitate a goal-directed programming task.

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 [9, 3, 11, 8, 2]. RL techniques have begun to show promising results for program synthesis [7, 10, 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 [12] or other user interface to explore the space of valid programs. Adaption to settings where style, terminology and document structure vary remains a challenge. Prior approaches do not consider the scope or variety of artifacts in a software project. Some attempt to learn a graph [10] but only consider the local structure of source code and still require 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

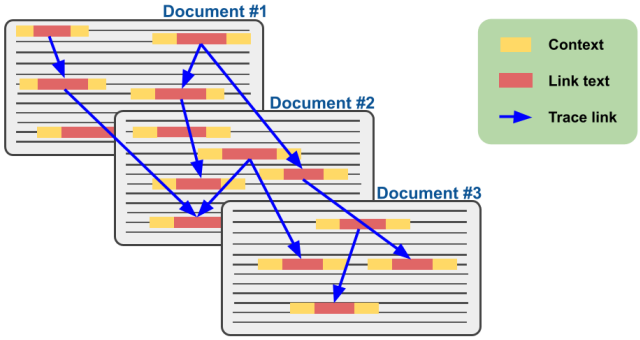

Figure 1: Random walk trace of locations visited by the document forager according to its traversal policy. These form a graph representing relevant entity links, which can be later used for prediction on a downstream task.

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

Consider a newly-hired developer, who has some programming experience, but no prior knowledge about a closed-source project. After onboarding, she gets access to the repository and is assigned her first ticket: Fix test broken by `0fb98be`. After finding the commit and getting familiar the code, she searches on StackOverflow, finds a relevant solution, copies some code into the editor, makes a few changes, presses run, and the test passes.

In order to accomplish a similar task, an information-seeking agent must be able to locate resources in a database to solve the task. Similar to a human developer, it must be able to perform simple actions, like `search(query)`, `read(text)`, `copy(text, from, to)`, and `runCode()`. Similar to navigating a physical environment, it can visit relevant documents and source code artifacts. As the agent explores the environment using these primitives, it builds up a project-specific knowledge graph. This graph can be later decoded at inference time to obtain the predicted solution.

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

2 Method

We fetch a dataset of random repositories on GitHub containing a mixture of filetypes representing both source code and natural language artifacts. Each project is indexed using a variable height radix tree, producing a multimap of prefix strings to a set of artifact, offset (A, O) pairs in linear time.

We initialize the policy network using a pretrained language model for the target languages. Starting at the site of the prediction task and conditioning on the context, the policy network draws K queries from its latent state. Querying the radix tree produces a set of matching locations within the project, and the process is repeated at each matching location using either a simple BFS traversal, or by expanding a subset of matching locations which are most promising, using MCTS with a finite horizon.

The rollout traces form a graph of related locations inside the project and their corresponding context embeddings, which become the GNN node features. Once the rollout ends, we run message passing on the resulting GNN for a fixed numbers of steps, then decode the graph embedding. 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 work seeks to answer the question: which artifacts in a software project are most useful for a given prediction task? We hypothesize that the policy network will learn to use both natural language and source code artifacts. If successful, this can be useful for both the prediction task itself, and knowledge graph extraction. Our research seeks to answer the following research questions. Can the agent can learn to:

1. Navigate the project and locate contextually relevant artifacts?
2. Comprehend the semantic content of contextually relevant artifacts?
3. 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?

In our second experiment, we try to measure the information contained in various filetypes by ablating results in various filetypes. For trajectories

which explore various filetypes such as Markdown or Java, what information gain do resources of these filetypes provide?

Our next steps are to build an RL environment which allows an agent to interact with a knowledge base and runtime environment. We will use an in-memory graph database to store and load project artifacts to and from the knowledge base. To model the agent, we will use the actions described above for navigating the environment, searching for text and reading documents, with a transformer-based internal state, pretrained on a corpus of projects in the same language. Finally, to evaluate the agent, we will generate problem instances and if necessary, use a JIT to rapidly validate generated code fragments.

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] 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.

[9] 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.

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

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

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