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

Breandan Considine, Xujie Si, Jin Guo

February 22, 2021

Humans are adept information foraging agents. We can quickly find relevant information in a large database, without need to read every document in detail. Instead, we use certain shortcuts to locate useful information. Documents like code and web pages contain many clues for how to read and write software. In this work, we will try to emulate the human's behavior when writing code and searching for programming-related documents.

Prior work focuses on building an index or learning an explicit mapping between concepts. This approach requires a lot of data and computation, and does not generalize across domains, where terminology and document structure may be very different. Instead, we will train an agent to solve a task by navigating a lazily-instantiated knowledge graph. The graph is too large to fit in memory, but links may be expanded by the agent on-demand.

The agent is like a newly-hired developer, who has experience writing code, but no prior knowledge of the closed-source project. After onboarding, she gets access to the code and is assigned her first ticket: Fix test broken by `0fb98be`. After finding the commit and geting familiar the codebase, she searches on StackOverflow and finds a relevant issue, then pastes some code into the IDE, changes a few variable names, presses run, and the test passes. "Well done!" says her boss, who gives her a high-five and a new ticket.

In order to accomplish some task such as fixing a test or finding a vulnerability, an information-seeking agent must locate resources in a knowledge graph to solve the task. To do so, it needs the ability to perform simple actions, like `readParagraph()`, `clickLink()`, `copyCode()`, `pasteCode()` `findText()` and `runCode()`. The agent has a timer (e.g. wall clock time or CPU cycles), and each action requires some time to complete. During the allotted time, the agent can explore the graph or try various solutions.

As the agent is exploring the knowledge graph, it collects information about how to solve the task. The agent might decide to read parts of doc-

uments and other source code artifacts. Similar to navigating a physical environment, the agent can choose to read a single document, or choose to jump around and visit multiple documents, clicking links or using the search bar. As the agent explores the documentation, they gather information in a latent state, which can be later decoded to the proposed solution.

Once the timer expires, we will then measure how well the agent performed. Many loss functions are possible depending on the assigned 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 property of the output) which must be satisfied by interacting with the runtime environment. Many program analysis and repair tasks are amenable to the described setup, including vulnerability or defect detection and correction.

Prior work in this area has focused on learning to synthesize programs [2] or navigate a graphical user interface [1], but does not consider the graphical structure of procedural knowledge. In this work, our primary focus is learning to read software artifacts in a software repository and apply the knowledge contained. Code is a graph database, or knowledge graph whose edges represent semantic relationships between entities. 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?

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 use a JIT to rapidly validate generated code fragments.

References

[1] gym-pc for OpenAI Gym. https://github.com/breandan/gym-pc, 2019.

[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.