Thoughts on how we Evaluate AI Assistants and the Python Coding Dataset

Ishan Gaur
EECS Dept.
UC Berkeley
`ishang@berkeley.edu`

June 1, 2025

1 Take-aways

One weakeness of the current approach is that the algorithmic complexity is quite low and there aren't many design tradeoffs for the user and assistant to consider.

Two key downsides of current assistants is that they try to do too much at once, which means your understanding of the problem and the solution lag far behind, and they don't help you discover a simpler problem/solution to solve that still meets your goals.

With our current setup, what if the reference implementation is just not how most people would interpret the problem–guess that is the query complexity of discovering the reference implementation.

2 Assistance Game Evaluation

Core to the idea of an assistance games[1] is that the user is the arbitrator of preferences and the assistant helps the user achieve those goals. In this work, we additionally consider that the user may not have a full operational definition of their preferences.[2].

How do we evaluate a coding assistant? The two key aspects are to see if it helps:

- the user effectively define their goal, for themselves and the assistant

[1] aka cooperative inverse reinforcement learning

[2] In general, a viable solution to maximize their preferences may not exist, such as a correct proof of a mathematical contradiction. In this sense, what we mean by "knowing" here is weaker than the sense of "knowing" the next prime larger than the largest known prime that Scott Aaronson discusses in [1]

- minimize the number of actions the user must take to meet the goal, including having to react to information that the assistant provides.

2.1 Defining a Goal

We don't want our RL algorithm to be rate-limited by collecting training episodes from interactions with real humans solving organic coding problems. Therefore, we sought to create a synthetic dataset and environment, to finally be used with real humans, that evoke the key aspects of the organic setting. These are that:

1. The overall problem can be easily understood.
2. The problem is difficult or at least time-consuming for the user to solve themselves.
3. It should be hard for the user to define the entire goal at once. This can be due to multiple reasons:
    - Such a description would be too long, itself being an intensive task
    - The user may not think it's necessary to define certain requirements upfront for the assistant
    - The user may not even be aware of these requirements until confronted with a situation in which they become apparent
    - The user may not have an exact implementation in mind, such as not having an exact algorithm or data-structure that they want the assistant to use

The first two items broadly filter the class of coding tasks that may be appropriate. For training assistants and evaluating them, part three is key.

These can be understood as:

1. Problem complexity, which is the complexity of the simplest valid solution to the vague problem statement
2. Specification complexity, which is the gap between
    - Minimum complexity across valid solutions
    - Implementation complexity of the optimal solution
3. Likelihood of specification discovery, which is the probability that the user will consider and discover all the relevant edge-cases and requirements of the desired behavior
4. Specification query complexity, which is the number of queries the user must make to fully discover the desired behavior once they are aware of a certain edge-cases

We want the assistant to help us bring down the problem and specification complexity if possible, while helping achieve the desired specification as fast as possible. The specification query complexity is a key problem in this. If it is too high then it will not be possible to discover the desired behavior in a reasonable amount of time. Additionally, it may be possible that the pair submit their solution without realizing that they did not discover the desired behavior due to low likelihood of specification discovery.

Desireable behaviors of the assistant. Help bring down the overall problem and specification complexity without reducing the overall reward. Being able to take on as much of the execution burden as possible. Reducing the overall time to achieve sufficiently high reward, in particular compared to the opportunity cost of doing more complex implementation or spending more time refining the problem.

- Predictability of assistant outcomes given user instructions
- The user's initial understanding of the problem and their preferences
- The user's final understanding of the problem and their preferences

2.2 Modeling Hidden Preferences

To model this, we allow the user to have query access to the reference implementation of the function they are trying to write. The idea is the user can try different inputs to figure out what the desired behavior is. The assistant can ideally even suggest inputs to try if it wants the user to clarify the desired behavior under certain conditions.

2.3 Dataset Filtering

There is a tension between the problem being easy to understand and being difficult enough to warrant the use of an assistant. After filtering for functions that can be run/evaluated outside of the context of the original code, we further needed to address this tension.

As a proxy for detecting problems that are too easy, we simply tested if GPT could solve the problem in a single step given the function signature and docstring.

Next we needed to test if the remaining tasks were too difficult. In particular, we wanted to test if it would be too hard to discover the desired behavior.

References

[1] Scott Aaronson and Alan M Turing. "10 Why Philosophers Should Care about Computational Complexity". In: *Computability: Turing, Gödel, Church, and Beyond* (2013), p. 261.