LLM + Logic Onboarding - Task 2

(Auditors - Sameera Sudarshan Kashyap and Pavithra Selvaraj) | Github

OpenAI documentation: Agentic AI

1. Agents are systems that are capable of creating complex workflows using a variety of component in domains like -
2. Models - used for decision making and reasoning
3. Tools - used to communicate with the outside world like function calling, web search etc.
4. Knowledge and memory - agents with external knowledge, eg: vector stores and embeddings
5. Audio and speech - used to create agents that are capable of understanding and responding back in natural language.
6. Guardrails - used to stop agents from exhibiting undesirable behaviour.
7. Orchestration - used in the development and deployment of agents.

Models such as o3 and o4 mini, versions of GPT 4.1 like mini and nano are capable of solving the difficult tasks, understand text, images, audio, code and documents and also supports real-time audio conversations.

Building voice agents (using OpenAI API and Agents SDK):

1. Choose one of the architecture for building voice agents:

Speech-to-speech - This is a multimodal architecture which directly processes audio and generates output in real time (gpt-4o-realtime-preview). This model takes into account the emotions, intent of the speaker and eliminates noise. This is mainly used for scenarios where interactive conversations and low latency are expected.

Chained architecture - This architecture processes the audio in a sequential manner - converts audio to text → generate response using LLM → synthesize audio from text which is recommended for a novice who is building voice agents. Unlike Speech-to-speech architecture, we have transcript in hand which can be tweaked to control our application

2. Building a voice agent:

Speech-to-speech architecture for real-time processing : Create a real time data transfer connection and create a real time session using Realtime API and use an OpenAI model with the capability of real time audio and input.

Chain together audio input, text processing and audio output: Install openAI agents SDK using the command “pip install openai-agents[voice]”. Run a speech-to-text model to convert audio to text, run the code to produce the result and finally run the text-to-speech model to turn the resulting text to audio.

Experiment using OpenAI + Prolog

Setup

- Setup the local environment by gathering the required tools
- Configured a new conda environment and installed packages pylog and openai sdk
- Installed `swi-prolog` tool using homebrew.
- Created a new github repository for AIEA tasks.
- Created and configured a new OpenAI API key from the dashboard into the .env local file.

Experiment

Prolog Generation using prompts

- **Prompt 1:** Utilized the sample BFS vs DFS prompt the generate baseline comparative facts about each approach

```python
text_prompt = "when is dfs better than bfs"
text_response = client.responses.create(
    model="gpt-4.1",
    input=text_prompt
).output_text
print(text_response)
```

- **Prompt 2:** From the output generated by the first prompt, we asked the model to generate equivalent prolog code.

```python
prolog_prompt = f"Turn this response into prolog - {text_response}"
prolog_response = client.responses.create(
    model="gpt-4.1",
    input=prolog_prompt
).output_text
```

```
print(prolog_response)
```

- Finally from the output of the prolog we wrote the python script using a prolog interface library to declare facts, rules and summaries and produce outputs for the query.
- Pylog did not work as mentioned in the documentation and we had to switch over `pyswip` to implement the program. While another issue was pyswip was not available in conda channels, and resorted to installing through `pip`

Sample code

```python
from pyswip import Prolog
prolog = Prolog()
# Query
for result in prolog.query("dfs_better(Scenario, Why, Example)"):
    print(result)
```

Final Output

```
{'Scenario': 'deep_solution', 'Why': 'DFS can reach deep solutions faster', 'Example': 'Puzzle game'}
{'Scenario': 'memory_constraints', 'Why': 'DFS uses less memory on wide graphs, since it only tracks the current path.', 'Example': 'Very wide trees or mazes'}
{'Scenario': 'any_solution', 'Why': 'DFS may quickly find some solution, without guaranteeing the shortest path.', 'Example': 'Maze generation; any valid maze is acceptable'}
{'Scenario': 'cycle_detection', 'Why': 'DFS can efficiently check for cycles by tracking the current path.', 'Example': 'Deadlock detection in operating systems'}
{'Scenario': 'strongly_connected_components', 'Why': 'DFS is the basis of Kosarajus and Tarjans algorithms for SCCs.', 'Example': 'Analyzing SCCs in social networks'}
{'Scenario': 'shortest_path', 'Why': 'BFS finds shortest path', 'Example': 'Maze solving'}
```

{'Scenario': 'level_order_traversal', 'Why': 'BFS explores layer by layer, so all nodes at one depth are explored before going deeper.', 'Example': 'Level-order traversal of trees'}
{'Scenario': 'shallow_solutions', 'Why': 'BFS is more efficient when all solutions are close to the root.', 'Example': 'Locating nearest exit in an emergency route plan'}