Shram

Name: Ashutosh Bharti
Github: https://github.com/Ashutosh212/persistent-memory-agent
Demo Video: Link

Technical Assignment: Long-Term Memory for LLMs

Introduction

This report details the development of a long-term memory agent with persistent storage, designed to support the addition, retrieval, and deletion of user-specific facts across sessions for Language Learning Models (LLMs).

Why is Long-Term Memory Important for Agentic Systems?

1. **Enhanced Coherence and Personalization:** Memory allows AI agents to learn from past interactions, retain context, and maintain consistency.
2. **Addressing Stateless LLMs:** OpenAI's API is stateless, meaning each request is independent. To maintain context, previous messages or relevant information must be explicitly passed.
3. **Mitigating Context Window Limits:** LLMs have a maximum input token limit. Passing the entire conversation history is not feasible.

Consequences of Not Incorporating Memory

4. **Frustrating User Experience:** Leads to repetitive questions and inconsistent behavior.
5. **Lack of Personalization:** The agent cannot adapt to individual user preferences.

Types of Memory

Short-term memory	Long-term memory
Works like a computer's RAM, holding relevant details for an ongoing task or conversation.	Works more like a hard drive, storing vast amounts of information to be accessed later.
Exists only briefly within a conversation thread.	Persists across multiple task runs or conversations.

Long term memories can be further divided into three types (CoALA framework paper):

1. **Episodic Memory:** Learns past events and experiences (e.g., past ticket booking).
2. **Procedural Memory:** Instruction(how to perform tasks and skills)
3. **Semantic Memory:** Stores general knowledge, facts, concepts

In this assignment, the focus is solely on storing the user's **Semantic Memory**. However, the system can be easily extended to incorporate episodic and procedural memory.

Challenges in Managing Long-Term Memory

Managing long-term memory is a complex task and remains a key research area due to challenges such as:

1. Determining which types of memories to store.
2. Figuring out what specific information to store.
3. Deciding on the optimal storage format.
4. Developing methods to decay older memories.
5. Effectively retrieving memories into working memory.

Efficient storage of memories is essential due to:

1. LLM context window limitations.
2. The risk of context pollution.

My Approach

Initial Setup

The project began with a first-principles approach using naive OpenAI calls. All user queries and responses were stored in a list, and the entire chat history was ingested as input for each API call. This allowed the LLM to recall previous conversations and user preferences.

User preferences are saved in a text file that is used before answering the user's current query.

1. `chat_history.txt` stores past conversations irrespective of the response.
2. `chat.completions.create()` generates a new response using the saved context.
3. Another `chat.completions.create()` call appends the current interaction to the file.

Subsequently, the shortcomings of this approach were analyzed to build a more robust system.

Problems with the Above Approach:

1. A flat text file is not structured memory. Also, GPT input context is limited, requiring careful selection of relevant memory.

2. Lack of semantic understanding of user preferences.
3. Retrieving all saved information without checking for similarity.
4. Not checking for duplicate memory
5. No deletion logic.

Second Step

An Agentic Workflow was built using LangGraph, where all chat history was added to storage. A node was defined to check if the user's query contained any personal information; if so, this information was saved. Otherwise, the workflow moved to a retrieval node to fetch any saved information before calling the model.(Agent WorkFlow Image)

Additions in this step:

1. Ability to update memory.
2. Checking if memory needs actual updating or if it already exists.
3. Retrieval of memory.

Still Shortcomings in the System:

1. Retrieval of the whole memory for a response.
2. No Vector Embedding logic for similar retrieval.
3. No structured way to store memory.
4. No logic for deletion.
5. The `InMemoryStore` in LangGraph stores data in RAM, making it volatile. A custom `MemoryStore()` was necessary.

Final Workflow

My Assumptions:

Every user query contains 0-N number of semantic information. Each will be extracted as a separate list.

Example:
Prompt: "I love hiking in the mountains and drinking cold brew coffee."

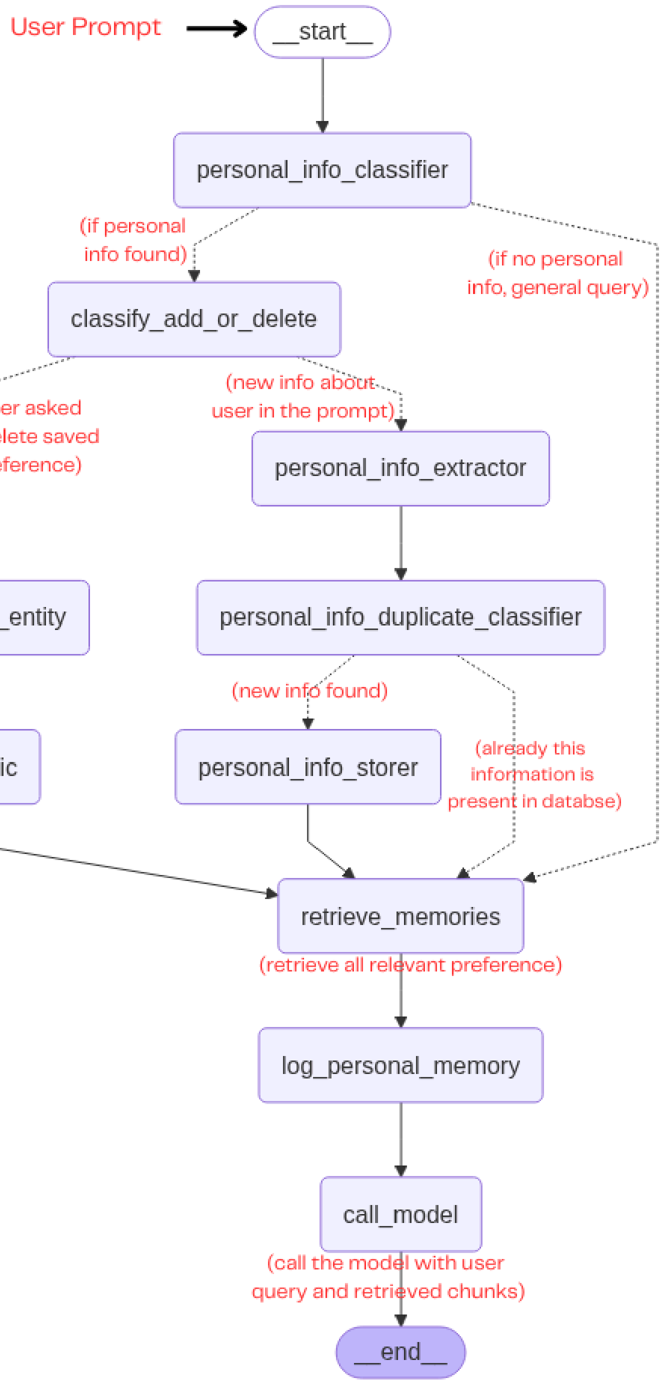
Saved Memory: `['hiking in the mountains', 'drinking cold brew coffee']`

Engineering in the final Agent Workflow for robustness:

1. Built the entire memory store logic from scratch for saving, retrieving, and deleting preferences from persistent storage. (Code)
2. Added deletion logic with a hybrid fusion of both embedding-based approach and regex matching. Quick experimentation determined a threshold of 0.6, above which two preferences are considered the same and removed.

3. Implemented a duplicacy check to ensure that user preferences are stored only if they are not already in the store.
4. The system is easily scalable to N number of users.

Final Workflow Agent:



Detailed explanation of what each node does is available in the Readme file of the project repository.

Results

The following achievements were made:

- ➔ **Storage:** Efficiently designing and implementing a memory store using a JSON-based schema and embeddings.
- ➔ **Detection:** Identifying storable information, such as personal facts and preferences, within user messages.
- ➔ **Extraction:** Extracting structured memory from unstructured text (e.g., recognizing "biryani and pizza" as a food preference).
- ➔ **Retrieval:** Retrieving relevant memories during new conversations based on user queries using cosine similarity between embeddings.
- ➔ **Deletion:** The agent can identify when a user requests the deletion of preferences, identify the relevant entities, and successfully delete them from storage using hybrid fusion.

Future Work and Improvement

Numerous improvements are possible:

1. **Improved Store Management:** Transition from JSON and NumPy formats to SQL databases for faster storage, deletion, and updates, improving overall time complexity.
2. **Enhanced Routing Algorithm:** The current binary model either deletes or adds preferences. A more sophisticated model is needed to handle queries requiring both operations (e.g., "Instead of Notion, I like to use Shram" necessitates deleting previous information and adding new).
3. **Negative Preference Handling:** The system currently does not save negative preferences or respond to commands like "remove all my preferences."
4. **Faster Retrieval:** Currently, cosine similarity is calculated with every stored embedding, which is slow for large datasets. Faster algorithms like ANN (Approximate Nearest Neighbor) or vector databases like Weaviate could significantly improve latency.
5. **Prompt Engineering:** The current use of few-shot prompting can be improved with other system prompts to enhance LLM accuracy and reasoning capabilities.
6. **LLM as Query Generator:** Using the LLM to make decisions about when to retrieve long-term memory, generating queries for searches (via function calling tokens), and then using vector search for relevant chunks.
7. **Thresholding Extraction:** The current limit-based approach extracts a fixed number of top preferences. A thresholding-based approach, dynamically retrieving chunks from the database based on proper experimentation, could be more effective.

Conclusion

This project addresses a very interesting and actively researched problem that requires careful consideration at every step. The engineered system is capable of persisting a user's long-term memory by saving their preferences. This capability allows for a more personalized and seamless user experience, as the system can adapt to individual needs and behaviors over time.