PROJECT OVERVIEW

ShopAssist AI 2.0

This document details the architecture and implementation of ShopAssist AI 2.0, a chatbot designed to help users find suitable laptops. This version leverages OpenAI's function calling API for enhanced interactivity and accuracy, marking a significant improvement over the previous iteration.

Project Overview

ShopAssist AI 2.0 helps users find laptops based on their needs and preferences. It uses a conversational approach to gather user requirements, including budget, desired features (GPU, display, portability, multitasking, processing speed), and potential user personas (gamer, student, etc.). The chatbot then utilizes these criteria to filter and recommend relevant laptops from a provided dataset. The core enhancement in version 2.0 is the integration of OpenAI's function calling API, which enables the chatbot to directly interact with dedicated Python functions, making the process more robust and efficient.

System Architecture

ShopAssist AI 2.0 utilizes a modular architecture, dividing its functionality into five key Python files:

1. config.py: This file serves as the central repository for all configuration parameters, ensuring easy management and modification. It stores:
    - API Keys: Holds the OpenAI API key, loaded securely from a .env file.
    - File Paths: Specifies the locations of input and output data files (raw laptop data, preprocessed data).
    - Model IDs: Defines the specific OpenAI models used for both the core chatbot logic and the data preprocessing tasks. This allows for flexibility in choosing different models based on performance and cost considerations.
    - Data Processing Flags: Controls aspects of data preprocessing, such as whether to force reprocessing or use cached data.
    - Chatbot Settings: Sets parameters related to the chatbot's behavior, like the maximum conversation history length.
    - Persona Values: Maintains a list of available user personas, ensuring consistency throughout the system and allowing easy customization.
    - Prompt Delimiters: Defines special markers used within LLM prompts for clarity and parsing.

2. llm_utils.py: This module encapsulates all interactions with the OpenAI API, providing abstraction and simplifying the code in other modules. Key functions include:
    - get_chat_completions_for_preprocessing: A specialized wrapper for OpenAI chat completions, optimized for data preprocessing tasks. It utilizes a low temperature setting to ensure deterministic output when extracting structured information (JSON) from the LLM. It includes robust error handling for JSON parsing and unexpected LLM outputs, attempting to extract correct JSON even when embedded in markdown or other characters.
    - get_chatbot_completion: A wrapper function for OpenAI chat completions, tailored for the main chatbot conversation loop. It accepts messages, optional function call definitions (tools), and a model ID, returning the LLM's response. This function also handles potential errors during API calls, providing a fallback message to maintain conversation flow.
3. laptop_data_manager.py: This module manages the entire lifecycle of laptop data, from loading and preprocessing to providing access to the processed data. Its responsibilities include:
    - Data Loading: Loads the raw laptop data from the specified CSV file. Includes error handling for missing files and incorrect CSV formats.
    - Preprocessing: Implements the data preprocessing pipeline using LLM helper functions:

        - _product_map_layer: Classifies laptop features (GPU intensity, display quality, etc.) from product descriptions using an LLM, categorizing them into levels (low, medium, high). Includes robust prompt engineering and error handling for unexpected LLM outputs.
        - _persona_tag: Assigns relevant user personas to each laptop based on its classified features, utilizing a structured prompt and JSON extraction. Includes error handling for inconsistencies or invalid input specifications.
    - Data Caching: Caches the preprocessed data to a separate CSV file, significantly improving performance on subsequent runs by avoiding redundant LLM calls.
    - Data Access: Provides a function get_laptop_dataframe to allow other modules to access the loaded and preprocessed data as a Pandas DataFrame. Handles cases where the data has not yet been initialized. Includes improved handling of stringified JSON data during loading.
4. chatbot_functions.py: This module defines the "tool" functions that are accessible to the LLM through the function calling API. These functions provide the core functionality of the chatbot:
    - get_laptop_info: Retrieves detailed information about a specific laptop model by name. It performs case-insensitive searches, handles missing data gracefully, and returns the information in JSON format.
    - recommend_laptops_by_criteria: Recommends laptops based on user-provided criteria (budget and persona). It filters the laptop data based on these criteria, returning the top matching laptops in JSON format.

The filtering logic has been improved to handle edge cases and inconsistencies, providing more relevant results.

- end_conversation: Signals the end of the conversation, providing a goodbye message.
- get_available_functions_map: Returns a dictionary mapping function names to the actual function objects, making it easy for main_chatbot to execute the correct function based on the LLM's tool_call.
- get_tools_definition: Generates the tool definition JSON required by the OpenAI API, including function descriptions, parameters, and required arguments. This definition is dynamic, incorporating the PERSONA_VALUES from config.py, making it flexible and maintainable.

5. main_chatbot.py: This module orchestrates the entire chatbot interaction, managing the conversation flow and executing tool functions. Key features include:
   - Initialization: Initializes the laptop_data_manager to load and preprocess the data.
   - System Prompt: Defines the initial system prompt for the LLM, which guides the chatbot's behavior and explains the conversational flow, the available functions, and persona options.
   - Conversation Loop: Implements the main conversation loop, which:
     - Gets user input.
     - Sends user input and history to the LLM.
     - Handles LLM responses, including function calls:

        - Identifies and executes called tool functions.
        - Formats and presents function outputs to the user.
    - Manages conversation history and pruning.
    - Handles exceptions and errors.
- Error Handling and Recovery: Includes comprehensive error handling for various scenarios, including API errors, JSON parsing issues, function call exceptions, and unexpected LLM outputs. Provides helpful messages to the user and recovery mechanisms to keep the conversation flowing.

Function Calling API Integration

The function calling API is integrated into main_chatbot.py. Within the main loop, the LLM response is checked for tool_calls. If present, the script iterates through the tool calls, identifies the function to be executed from chatbot_functions.py using a function map, prepares the arguments, and executes the function. The function's JSON output is then incorporated back into the conversation history, allowing the LLM to provide a relevant user-facing response based on the function's output.

Modifications to Existing System

The key changes from the ShopAssist AI 1.0 to the new modular structure are:

1. **Modularization:** The code is now organized into separate, well-defined modules (config, llm_utils, laptop_data_manager, chatbot_functions, and

main_chatbot). This improves code maintainability, readability, and testability compared to the single notebook file approach in the old code.

2. **Function Calling:** The central change is the use of OpenAI's function calling. This replaces the old logic of manually parsing LLM responses for information like user requirements and laptop features, significantly improving accuracy and reliability.
3. **Data Preprocessing:** Data preprocessing is now handled in a dedicated module (laptop_data_manager), with improved handling of missing/invalid values and a structured prompt for classification. The old code lacked this structure and clear error management.
4. **Robustness:** Extensive error handling is added throughout the new code to address issues like invalid JSON, missing data, and API errors. The old code had minimal error handling, making it prone to unexpected failures.
5. **Data Caching:** The laptop_data_manager module caches preprocessed data to avoid repeating costly LLM calls, leading to faster performance after the initial run.
6. **Dynamic Personas:** The persona list for function calling is dynamically generated from config.py, making it easily customizable.
7. **Concise Recommendations:** The new recommend_laptops_by_criteria function returns a more focused subset of columns to make recommendations clearer for the user, unlike the old code that might have included irrelevant data.
8. **Robust Filtering and Persona Matching:** The code now handles the persona filtering and matching process correctly. It performs a case-

insensitive match and ensures that the persona column is properly checked in the dataframe.

9. **Improved API Interactions:** The llm_utils module provides structured wrappers for OpenAI API calls, handling both preprocessing functions (with lower temperature for JSON extraction) and chatbot interactions separately. It also checks the API response for unexpected formats and attempts to extract the correct JSON even if there are markdown or other characters.

User Experience Enhancements

The integration of the function calling API and the modular design result in several user experience improvements:

- **More Relevant Recommendations:** By leveraging structured data and function calls, the chatbot provides more accurate and targeted laptop recommendations based on user preferences. The refined filtering logic and persona matching ensure that users are presented with the most suitable options.
- **Enhanced Interactivity:** The chatbot now engages in a more structured and dynamic conversation, guiding users through the process of specifying their requirements and providing tailored responses based on function outputs.
- **Improved Clarity and Conciseness:** Laptop recommendations are presented with relevant details, avoiding information overload and improving the readability of the responses.

- **Robustness and Error Handling:** The improved error handling and recovery mechanisms ensure a smoother user experience, minimizing interruptions due to unexpected issues. The chatbot gracefully handles invalid input or unexpected data, providing helpful messages and prompting users for clarification.
- **Faster Responses (after initial loading):** Data caching reduces the latency for retrieving laptop information, leading to faster responses and a more interactive conversation.

Code Walkthrough / Highlights

The core enhancements in the new code are best illustrated through a walkthrough of how it handles a user request:

1. User Input: The user provides input, for example, "I'm a gamer looking for a laptop under 100000."
2. LLM Interaction: main_chatbot.py sends this input to the LLM via llm_utils.get_chatbot_completion. The LLM, guided by the system prompt, recognizes the user's intent (laptop recommendation) and the provided criteria (gamer persona, budget).
3. Function Call: The LLM decides to call the recommend_laptops_by_criteria function, formulating a tool_call with the arguments {"budget_max": 100000, "personas": ["gamer"]}.

4. Tool Execution: main_chatbot.py identifies this tool_call, retrieves the corresponding function from chatbot_functions.get_available_functions_map, and executes it with the provided arguments.
5. Data Retrieval: recommend_laptops_by_criteria accesses the preprocessed laptop data through laptop_data_manager.get_laptop_dataframe. This step benefits from caching if the data has been processed previously.
6. Filtering and Recommendation: The function filters the DataFrame based on budget and persona. The enhanced filtering logic ensures more accurate matches by considering potentially incomplete or unstructured persona lists associated with laptops.
7. JSON Output: The function returns the top matching laptops as a JSON string.
8. LLM Response Generation: The JSON result is added back into the message history. The LLM receives this output and uses it to generate a human-friendly response for the user, like: "I found 3 laptops matching your criteria: [Laptop 1 details], [Laptop 2 details], [Laptop 3 details]. Which one are you most interested in?"
9. Conversation Continues: The chatbot continues the interaction, potentially answering further user questions about specific laptops using the get_laptop_info function or other tool functions as needed.

Conclusion and Benefits

ShopAssist AI 2.0 demonstrates a substantial advancement in chatbot design for product recommendation by integrating OpenAI's function calling API and adopting a modular architecture. The key benefits include:

- Increased Accuracy and Relevance: Function calling enables precise interaction with structured data, eliminating the ambiguity and potential errors of relying solely on LLM parsing for information extraction. This leads to more accurate filtering, scoring, and ultimately, more relevant recommendations.
- Improved User Experience: The structured conversation flow, guided by the LLM and tool functions, makes the interaction more intuitive and efficient for users. Clear, concise recommendations and robust error handling further enhance the user experience.
- Enhanced Maintainability and Scalability: The modular design significantly improves code organization and readability, making it easier to maintain, debug, and extend the system. Adding new tool functions or data sources becomes a more straightforward process.
- Cost and Performance Optimization: Caching preprocessed data reduces the number of API calls required, minimizing costs and improving response times after the initial data loading.
- Flexibility in Model Selection: Decoupling model IDs in the configuration allows for easy experimentation with different LLM models for different parts of the system (e.g., a powerful model for the chatbot logic and a potentially less expensive model for data preprocessing).

ShopAssist AI 2.0 demonstrates the potential of function calling to transform chatbot interactions, making them more intelligent, reliable, and user-friendly. The robust and scalable architecture positions the system for future enhancements and demonstrates best practices in LLM-powered application design.