

Fashion Assistant Chatbot Documentation

The **Fashion Assistant Chatbot** helps users find fashion products based on their queries. This project integrates several key technologies, including OpenAI's GPT-4, ChromaDB, and embeddings, to provide personalized recommendations. The chatbot supports both new searches and follow-up queries, with the ability to maintain context for a more seamless user experience.

The project consists of three core layers — **Embedding Layer**, **Search Layer**, and **Generation Layer** — and includes a memory-enabled version to enhance the chatbot's interaction with users.

Data: [Myntra Fashion Product Dataset](#)

Steps Taken to Complete the Fashion Assistant Chatbot Project

Step 1: Project Setup and Configuration

1. API Key Management:

- **Method:** The API keys were securely stored in a configuration file (`config.yaml`), and the `load_yaml(file_path)` function was used to read these keys.
- **Details:** The OpenAI API key is loaded and set as an environment variable for safe usage in the embedding and generation processes.

2. Environment Setup:

- **Method:** Installed required libraries such as `chromadb`, `openai`, `pandas`, `sentence-transformers`, and `yaml`.
 - **Details:** These libraries enable handling embeddings, managing the database, querying, and generating responses using GPT-4.
-

Step 2: Embedding Layer Implementation

1. Data Preprocessing:

- **Method:** Read the fashion product dataset (`Fashion Dataset v2.csv`) and split it into smaller chunks to facilitate processing.
- **Details:** Used the `chunk_data()` function to manage large datasets by splitting them into smaller portions, which were then prepared for embedding.

2. Data Transformation:

- **Method:** Combine product attributes such as name, brand, color, price, and description into a single string for embedding.
 - **Details:** This combined string allowed for the generation of embeddings, while metadata was created to store additional product details for later use.
3. **Embedding the Data:**
- **Method:** Used OpenAI's `text-embedding-ada-002` model to generate embeddings for each product.
 - **Details:** These embeddings were stored in ChromaDB, creating a persistent collection of vectorized product data for fast retrieval.
-

Step 3: Search Layer Implementation

1. **Embedding Storage and Querying:**

- **Method:** Set up ChromaDB to store embeddings and metadata. The `query_collection()` function was used to search the database for relevant products based on user queries.
- **Details:** The query process used semantic similarity to match user queries with stored product embeddings.

2. **Moderation and Content Filtering:**

- **Method:** Implemented content moderation using OpenAI's moderation API to filter inappropriate user input.
- **Details:** The `check_moderation()` function ensures safe interactions by flagging inappropriate content.

3. **Reranking the Results:**

- **Method:** After retrieving results, the `rerank_results()` function used a cross-encoder to rerank the results based on their relevance to the user's query.
 - **Details:** This step helped prioritize the most relevant products, improving the quality of the suggestions.
-

Step 4: Generation Layer Implementation

1. **Formatting and Structuring Responses:**

- **Method:** Formatted the search results into a readable structure, including key product details like name, price, and description.
- **Details:** This was done within the `generate_response()` function, which organized the results into a structured response.

2. Response Generation using GPT-4:

- **Method:** Used OpenAI's GPT-4 model to generate a natural language response based on the top search results.
- **Details:** The `generate_response()` function sends formatted results and the query to GPT-4, which generates detailed conversational responses.

3. User Interaction via `chatbot()`:

- **Method:** Implemented the main function (`chatbot()`) to handle user interactions, check for moderation, perform searches, and display responses.
 - **Details:** This function supports both new searches and follow-up queries, ensuring smooth conversation flow.
-

Step 5: Memory-Enabled Chatbot Implementation (Optional)

1. Context Management:

- **Method:** Implemented memory functionality to handle follow-up queries by retaining context from previous interactions.
- **Details:** The `current_context` variable tracks the search history, ensuring relevant follow-up responses based on prior queries.

2. Contextual Response Generation:

- **Method:** For follow-up queries, the chatbot uses retained context to generate responses that refer to previous interactions.
 - **Details:** This was achieved by passing the query and the current context to GPT-4 for generating the appropriate response.
-

Step 6: Final Testing and Evaluation

1. Test Collection Retrieval:

- **Method:** Ran tests on the ChromaDB collection to ensure correct retrieval of relevant products based on semantic search.
- **Details:** Used the `test_collection()` method for testing before implementing advanced reranking features.

2. Query Testing and Validation:

- **Method:** Validated that the generated responses matched expectations for different queries.

- **Details:** Ensured that the products returned were relevant, and the responses were structured correctly by GPT-4.
-

Step 7: Deployment and Optimization

1. Optimizing Response Quality:

- **Method:** Adjusted the temperature in GPT-4 to 0.4 for better coherence in responses.
- **Details:** This setting strikes a balance between structured and creative responses, ensuring a natural but accurate reply.

2. Refining User Experience:

- **Method:** Added features like the "quit" and "new" commands to improve user interaction.
 - **Details:** These features allowed users to exit the chatbot gracefully or start a fresh search whenever needed.
-

Fashion Assistant Chatbot: Layer-wise Breakdown

1. Embedding Layer: `embedding.py`

The **Embedding Layer** handles data preprocessing and generates embeddings for fashion product data, which are stored for future use.

Key Methods:

- `load_yaml(file_path)`: Loads the configuration file containing API keys.
 - `chunk_data(main_csv_path, chunk_size=1000)`: Divides the dataset into smaller chunks for efficient processing.
 - `prepare_data(chunk)`: Combines relevant product details into a string for embedding.
 - `embedding_and_storing_data(fashion_df)`: Generates embeddings and stores them in ChromaDB.
-

2. Search Layer: `search_and_generate.py`

The **Search Layer** is responsible for querying the ChromaDB collection and retrieving relevant results.

Key Methods:

- **check_moderation(text)**: Ensures that the user's input is appropriate.
 - **query_collection(query)**: Queries the ChromaDB collection for relevant products.
 - **process_results(results)**: Processes the raw results into a structured DataFrame.
 - **rerank_results(query, results_df)**: Reranks the results to prioritize the most relevant products.
 - **generate_response(query, top_5_RAG)**: Generates the final response using GPT-4.
-

3. Generation Layer: **FashionChatbot.py**

The **Generation Layer** formats the search results and generates a response based on user queries using GPT-4.

Key Methods:

- **check_moderation(text)**: Checks for inappropriate content in user input.
 - **query_collection(query)**: Queries the ChromaDB collection.
 - **process_results(results)**: Processes results for response generation.
 - **rerank_results(query, results_df)**: Reranks the results for better relevance.
 - **generate_response(query, top_5_RAG)**: Generates a structured response using GPT-4.
-

Screenshot and Results Storage


Since the results can be large, they are stored in a text file instead of displaying in a screenshot. Below are the results of recent queries:

1. **Query 1:** *Cotton top for casual wear*
SubQuery: *price under 1500*

2. **Query 2:** *Rayon material kurtha*
SubQuery: *rating greater than 4.5*

3. **Query 3:** *blue color skirt under 2000*

The full results are stored in the corresponding text file for easy reference.

 Output

Conclusion

The **Fashion Assistant Chatbot** integrates embeddings, semantic search, and GPT-4 for natural language generation, providing an effective solution for fashion product discovery. The memory-enabled version enhances the user experience by maintaining context across interactions, making it suitable for more complex, ongoing conversations. This documentation outlines the steps and methods followed to build the chatbot, ensuring a robust, interactive, and user-friendly system.