Overall Approach

1. **Corpus Preparation**: Structure the provided corpus in a way that the chatbot can easily retrieve answers.
2. **Model Selection**: Use a pre-trained language model (e.g., OpenAI's GPT-3.5) to handle user queries and provide responses based on the corpus.
3. **Query Handling**: Implement logic to differentiate between in-corpus and out-of-corpus queries.
4. **Conversation History**: Maintain conversation context to handle follow-up questions.
5. **User Interface**: Develop a minimalistic web UI for user interaction.
6. **Deployment**: Deploy the chatbot on a web server.

Frameworks/Libraries/Tools

1. **Python**: Main programming language.
2. **Flask**: For creating the web server and API.
3. **HTML/CSS/JavaScript**: For the web UI.
4. **OpenAI GPT-3.5**: Pre-trained language model for generating responses.
5. **SQLite**: To store the corpus data.
6. **LangChain**: For maintaining conversation history and context management.

Implementation Steps

1. **Set Up the Environment**
   - Install necessary libraries:

bash

Copy code

```
pip install flask openai langchain sqlite3
```

2. **Corpus Preparation**
   - Convert the corpus into a structured format (e.g., JSON or SQLite database).
3. **Flask API**
   - Create an API endpoint that handles user queries.
   - Use OpenAI's GPT-3.5 to generate responses based on the corpus.
   - Implement logic to check if a query can be answered from the corpus.
4. **Web UI**
   - Create a minimalistic UI using HTML/CSS/JavaScript.
   - Implement AJAX calls to interact with the Flask API.

5. **Conversation History**
    - Use LangChain or custom logic to maintain conversation context and handle follow-up questions.
6. **Deployment**
    - Deploy the Flask app on a web server (e.g., Heroku, AWS).

Overall Approach

The goal was to create a chatbot that can answer user queries based on a specific corpus related to a wine business. The chatbot should:

1. **Retrieve Answers from Corpus**: Use the provided corpus to answer user questions accurately.
2. **Handle Out-of-Corpus Queries**: Direct users to contact the business if their queries are outside the scope of the corpus.
3. **Maintain Conversation Context**: Support follow-up questions by maintaining the context of the conversation.
4. **Provide a Minimalistic UI**: Offer a simple and user-friendly interface for interaction.

Steps Involved:

1. **Corpus Preparation**: The corpus was structured in a SQLite database, ensuring quick and efficient retrieval of information.
2. **Model Integration**: Integrated OpenAI's GPT-3.5 model for generating responses.
3. **Query Handling Logic**: Implemented a mechanism to differentiate between in-corpus and out-of-corpus queries.
4. **UI Development**: Created a minimalistic web interface using HTML, CSS, and JavaScript.
5. **Deployment**: Deployed the chatbot using Flask, a lightweight WSGI web application framework.

Frameworks/Libraries/Tools Used

1. **Python**: Primary language for backend development.
2. **Flask**: Used to create the web server and API endpoints.
3. **SQLite**: Database to store the corpus for efficient query handling.
4. **OpenAI GPT-3.5**: Language model used to generate responses.
5. **LangChain**: For maintaining conversation history and context management.
6. **HTML/CSS/JavaScript**: For creating the minimalistic user interface.
7. **AJAX**: To enable asynchronous communication between the frontend and backend.
8. **GitHub**: For version control and code repository.

Problems Faced and Solutions

1. **Corpus Structuring**:
   - **Problem**: Ensuring the corpus was structured in a way that allowed efficient query handling.
   - **Solution**: Used SQLite database to store the corpus, which provided quick retrieval and easy integration with the Flask application.
2. **Handling Follow-Up Questions**:
   - **Problem**: Maintaining conversation context to understand follow-up questions.
   - **Solution**: Implemented LangChain to manage conversation history and context, enabling the chatbot to interpret follow-up questions accurately.
3. **Latency Issues**:
   - **Problem**: Ensuring the chatbot responded within the acceptable latency limit.
   - **Solution**: Optimized the API calls to the GPT-3.5 model, implemented caching mechanisms, and fine-tuned the model for the specific corpus to reduce token usage and response time.
4. **Out-of-Corpus Queries**:
   - **Problem**: Differentiating between in-corpus and out-of-corpus queries.
   - **Solution**: Implemented logic to check if a query could be answered from the corpus. If not, the chatbot prompts users to contact the business directly.

Future Scope

1. **Enhanced UI/UX**:
   - **Objective**: Improve the web interface for a better user experience.
   - **Details**: Add more interactive elements, improve the design aesthetics, and ensure the UI is responsive and accessible on different devices.
2. **Multilingual Support**:
   - **Objective**: Add support for multiple languages to cater to a broader audience.
   - **Details**: Integrate language translation services and train the model to handle queries in different languages.
3. **Advanced Analytics**:
   - **Objective**: Implement analytics to track user interactions and improve responses.
   - **Details**: Use tools like Google Analytics to monitor user behavior, gather insights, and refine the chatbot's performance based on user data.
4. **Personalization**:
   - **Objective**: Customize responses based on user preferences.

    - **Details**: Implement user profiling and preference storage to provide personalized recommendations and responses.

Token Usage Optimization

1. **Caching Mechanisms**:
    - **Objective**: Reduce redundant API calls to minimize token usage.
    - **Details**: Cache frequent queries and their responses. Implement a mechanism to check the cache before making API calls.
2. **Fine-Tuning the Model**:
    - **Objective**: Reduce token usage by optimizing the model for the specific corpus.
    - **Details**: Fine-tune the GPT-3.5 model using the specific corpus data to make it more efficient in generating relevant responses, thereby reducing the number of tokens used per query.

README Instructions

1. **Run Flask Server**

```
python app.py
```

2. **Open Web UI**
    - Open index.html in a web browser.