Chatbot

Overview

1. Project Overview

The project aimed to develop a chatbot capable of processing, storing, and retrieving documents based on semantic content. It utilized Flask for the web framework, Weaviate as a vector database for semantic search, and OpenAI's GPT models for generating conversational responses. The chatbot was designed to handle two main functionalities: direct semantic queries and conversational queries that leverage document retrieval.

2. Development Process

Design and Planning

- **Technology Selection**: Chose Flask for its simplicity and Python compatibility, Weaviate for its vector search capabilities, and OpenAI for its advanced NLP models.
- **System Architecture**: Designed a modular architecture using Flask blueprints for scalability and organized codebase, and integrated Weaviate for data storage and retrieval.

Implementation

- **PDF Document Processing**: Implemented document loading and text extraction from PDFs to handle input data.
- **Text Embedding**: Utilized `sentence_transformers` for generating high-quality embeddings of document text and queries.
- **Database Schema**: Defined a Weaviate schema for storing documents with their embeddings.
- **API Integration**: Configured the OpenAI API for conversational responses and embedding generation.
- **Web Server Setup**: Developed Flask routes for handling semantic and conversational queries, including error handling and response formatting.

3. Challenges Encountered

Integration Complexities

- **Challenge**: Integrating Weaviate with Flask and ensuring efficient data retrieval.
- **Solution**: Thoroughly reviewed Weaviate documentation and utilized its Python client for seamless integration.

Model Selection and Performance

- **Challenge**: Selecting the right model for text embeddings and conversational responses.
- **Solution**: Evaluated various models and settled on `sentence_transformers` and OpenAI's GPT due to their performance and ease of integration.

Error Handling

- **Challenge**: Gracefully handling errors from API calls and database queries.
- **Solution**: Implemented robust error handling within Flask routes to ensure the application remains responsive.

4. Performance Evaluation

Semantic Query Handling

- **Evaluation**: Tested the chatbot's ability to accurately retrieve documents based on semantic queries.
- **Results**: Generally high accuracy in finding relevant documents, though performance varied with query specificity.

Conversational Responses

- **Evaluation**: Assessed the relevance and coherence of conversational responses generated by the OpenAI model.
- **Results**: Responses were contextually appropriate and informative, with occasional lapses in relevance due to the complexity of some queries.

Scalability and Response Time

- **Evaluation**: Monitored the application's response time and scalability under load.
- **Results**: Response times were acceptable, though scalability testing indicated potential bottlenecks in handling simultaneous requests, suggesting a need for optimization.

5. Lessons Learned and Future Directions

Continuous Learning

- The integration of advanced AI and database technologies requires ongoing learning and adaptation, particularly as new models and versions are released.

Optimization

- Future work will focus on optimizing the application's architecture for scalability and improving response times, possibly by caching frequent queries and responses.

Model Tuning

- Further tuning of the models used for embeddings and conversational responses could enhance accuracy and relevance.

6. Conclusion

The development of this chatbot demonstrated the feasibility and challenges of creating a semantic document processing and retrieval application using Flask, Weaviate, and OpenAI. While the chatbot performed well in semantic query handling and generating conversational responses, there is room for improvement in scalability, response time, and the precision of responses. The project highlighted the importance of careful technology selection, robust error handling, and the potential of AI to transform information retrieval and interaction.

Technical

Creating Vector Store

PDF Document Loading

```python
from langchain.document_loaders import PyPDFLoader

loader = PyPDFLoader("data_src.pdf")
pages = loader.load()
print(len(pages))
```

- **Design Decision**: Uses `PyPDFLoader` from `langchain.document_loaders` to load PDF documents.
- **Justification**: This loader abstracts the complexities of reading and parsing PDF files, allowing the code to work with a higher level representation of the document pages.

Text Splitting

```python
from langchain.text_splitter import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
  separator="\n",
  chunk_size=1000,
  chunk_overlap=150,
  length_function=len
)
docs = text_splitter.split_documents(pages)
print(len(docs))
```

- **Design Decision**: Implements `CharacterTextSplitter` for splitting document text into manageable chunks based on characters.
- **Justification**: This approach allows for fine-grained control over the size of text chunks, which is useful for processing large documents. The `chunk_overlap` parameter ensures that the context is preserved between adjacent chunks, which can be crucial for understanding and embeddings.

OpenAI API Integration

```python
import openai

API_KEY = ""
```

```
openai.api_key = API_KEY
os.environ["OPENAI_API_KEY"] = API_KEY
embedding = OpenAIEmbeddings()
```

- **Design Decision**: Configures OpenAI API access by setting the API key both in the `openai` library and as an environment variable.
- **Justification**: This ensures that any part of the code or underlying libraries that require access to OpenAI services can authenticate seamlessly.

Embedding Text

- **Design Decision**: Utilizes `OpenAIEmbeddings` to convert text into vector representations.
- **Justification**: Embedding the text allows for semantic analysis and comparison of documents, enabling functionalities like semantic search and similarity detection.

Vector Database Integration

```
from weaviate import Client, schema

client = Client("http://localhost:8080")
client.schema.delete_all()
client.schema.create_class(class_obj)
```

- **Design Decision**: Uses Weaviate, a vector database, for storing and querying document embeddings.
- **Justification**: Vector databases are optimized for storing high-dimensional data and performing nearest neighbor searches, making Weaviate an ideal choice for semantic search applications.

Document Insertion and Query

```
def insert_document(title, content, vector):
  document = {
    "title": title,
    "content": content
  }
  client.data_object.create(document, class_name="Document",
vector=vector)

for i in range(len(docs)):
  embeddings.append(embedding.embed_query(docs[i].page_content))
  insert_document(title, docs[i].page_content, embeddings[i])
```

- **Design Decision**: Defines `insert_document` function for adding documents to Weaviate with their embeddings.
- **Justification**: Abstracting the insertion logic into a function simplifies the process of storing documents along with their semantic representations, enabling efficient retrieval based on content similarity.

Semantic Query Execution

```
query_embedding = embedding.embed_query(query)
results = client.query.get(
  "Document",
  ["content"]
).with_near_vector(
  vector_query
).with_limit(k).do()
```

- **Design Decision**: Embeds a query string and performs a semantic search in Weaviate to find similar documents.
- **Justification**: By embedding the query and using Weaviate's `with_near_vector` function, the code leverages semantic similarity to retrieve documents that are contextually related to the query, demonstrating an advanced search capability beyond keyword matching.

Backend Flask App Setup

This code integrates Flask for web server functionality, Weaviate for vector database operations, various NLP tools for document processing and embedding generation, and the OpenAI API for leveraging AI-based conversational models. It demonstrates a comprehensive approach to building a web-based application that processes, stores, and retrieves documents based on semantic content. The documentation below explains the components and design decisions.

Flask Web Server Setup

- **Flask Setup**: Initializes a Flask application and routes.
- **Blueprint Registration**: Uses Flask blueprints (`bp`) to organize and register routes, facilitating modular application development.
- **Design Decision**: Flask is chosen for its simplicity and flexibility in setting up web servers, and blueprints help in maintaining a clean separation of concerns.

Weaviate Vector Database Integration

- **Weaviate Client Initialization**: Sets up a connection to a Weaviate instance.
- **Schema Definition and Creation**: Defines a schema for storing documents with title and content fields, omitting vector property for manual handling.
- **Design Decision**: Weaviate is selected for its vector database capabilities, enabling semantic search and retrieval based on document embeddings.

Document Processing and Embedding

- **PDF Loading**: Utilizes `PyPDFLoader` to load PDF documents for processing.
- **Embedding Generation**: Implements `generate_embeddings` function using `sentence_transformers` for creating document embeddings.
- **Design Decision**: The choice of `sentence_transformers` provides high-quality embeddings that capture the semantic meaning of text, suitable for semantic search applications.

Document Insertion and Query Handling

- **Document Insertion**: Defines `insert_document` function to store documents and their embeddings in Weaviate.
- **Query Handling**: Implements two Flask routes (`/query` and `/retrievalquery`) to handle semantic queries and conversational queries, respectively.
- **Design Decision**: The system allows for complex interactions, including direct semantic queries and conversational queries that leverage both document retrieval and AI-driven responses.

AI Model Integration for Conversational Responses

- **OpenAI API Integration**: Configures and uses the OpenAI API for generating responses to queries based on the context provided by similar documents retrieved from Weaviate.
- **Conversational Chains**: Leverages `ConversationalRetrievalChain` and `RetrievalQA` from `langchain` to integrate conversational AI models with document retrieval, enriching responses with contextually relevant information.
- **Design Decision**: The integration of OpenAI models enables sophisticated conversational capabilities, allowing the application to provide informative and context-aware answers to user queries.

Error Handling and Response Formatting

- **Error Handling**: Includes try-except blocks in Flask routes to handle exceptions gracefully, ensuring the web server remains robust against errors.
- **Response Formatting**: Uses `jsonify` to format responses, facilitating easy integration with front-end applications or APIs.
- **Design Decision**: Ensuring robust error handling and using standard response formats are best practices for web development, enhancing the reliability and usability of the application.

Configuration and Environmental Variables

- **API Key Configuration**: Configures the OpenAI API key for authentication, ensuring secure access to AI models.
- **Environment Variables**: Sets up environment variables for further configuration and integration flexibility.
- **Design Decision**: Externalizing sensitive information and configuration details into environment variables follows security best practices and enhances application flexibility.