Chatbot with LLM and RAG in Action | by Bill Huang | Feb, 2024 | MediumOpen in appSign upSign inWriteSign upSign inChatbot with LLM and RAG in ActionBill Huang·Follow5 min read·Feb 28, 2024--1ListenShareIntroductionHello friends, in this article I will guild you through creating a cutting-edge chatbot for recommender with the power of LLM and with advanced retrieval technology to handle complex questions with ease.By the end, you'll learn how to:Build vector space for RAGBuild Chatbot power with LLM using LangChainMake chatbot's decison smarter with AgentLarge Language Models (LLMs)LLMs, such as GPT-4, Llama 2, Mixtral 8x7B, have revolutionized the way machines understand and interact with human language. Think of LLMs as the chatbot's brain, capable of deep understanding and generating human-like responses.Retrieval-Augmented Generation (RAG)RAG acts as a bridge, combining the creative response generation of models with the precision of retrieval-based methods, ensuring chatbots can source and integrate external information for more accurate responses.Traditional Chatbot Process vs. Chatbot + LLMTraditional chatbots often rely on scripted responses, limiting their flexibility. LLM integration expands their capabilities, enabling a dynamic and contextual conversation.Imagine traditional chatbots as librarians limited to their own knowledge, answering only from what they've learned.In contrast, LLM-enhanced chatbots are like librarians with access to every book in the whole library, providing wider-ranging and more detailed assistance.Chatbot vs. Chatbot + RAGChatbots without RAG are limited to responding based on their pre-existing knowledge, which can lead to incorrect or "hallucinated" answers when faced with unfamiliar questions. This means they might guess the answer, often resulting in unreliable responses.On the other hand, chatbots enhanced with RAG can access and use external information to answer questions. This capability allows them to provide more accurate and reliable responses, even for questions outside their initial training data. By integrating RAG, chatbots become more versatile and dependable for users seeking information.Imagine Chatbot + RAG are like librarians not only read all the book in the library, but in the middle of your conversation, can instantly fetch and read any relative info in everywhere to answer your questions.That's the power of RAG!Step-by-Step Code ExplanationNext, I will show you the proecss in buiding a book and movie recommender chatbot using Python, LangChain, embedding-based retrieval strategies, and OpenAI's GPT-3.5.The source code is here:GitHub - billpku/Chatbot_LLM_RAG_in_ActionContribute to billpku/Chatbot_LLM_RAG_in_Action development by creating an account on GitHub.github.comPreparation of the Corpus:The foundation of any RAG system is a robust and comprehensive corpus.For this chatbot, movie and book data were meticulously formatted into JSON, ensuring each entry had a unique ID, title, summary and etc. This standardized format facilitates efficient data retrieval and processing.My data were based on the the CMU's corpus:BookSummaryMovieSummaryBuilding the Vector Store for RetrievalTo enhance a chatbot's ability to search semantically similar content, we create embeddings for each item in the corpus. These embeddings are generated using sentence embedding models, such as the one from Hugging Face, and are indexed using FAISS for fast retrieval.Generating EmbeddingsFirst, we convert the content into embeddings using a sentence embedding model. This allows the chatbot to perform semantic similarity searches. Here's how to generate embeddings using the Hugging Face model:from langchain.embeddings import HuggingFaceEmbeddingsembeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")Storing and Retrieving EmbeddingsNext, we need a service like FAISS to store these embeddings and retrieve the most similar ones based on user input. FAISS simplifies the storage and retrieval process for both book and movie recommendations. Here's how to create, save, and load a FAISS store:from langchain.vectorstores import FAISS# Create FAISS store from documentvector_store = FAISS.from_documents(document, self.embeddings)# Save the spacevector_store.save_local(save_path)# Load the space, with embeddingsvector_store = FAISS.load_local(save_path, embeddings)Chatbot Logic and User Interaction:At its core, the chatbot is designed to interpret user queries, retrieve pertinent information, and generate responses that incorporate this information. This involves converting queries into embeddings, searching the corpus, and then using GPT-3.5 to craft a final response that integrates both the retrieved data and the model's generative output.Using GPT-3.5 for Dynamic ResponsesWe employ GPT-3.5 to power our chatbot, enabling it to anticipate user needs and craft answers using our curated content.import osfrom langchain.chat_models import ChatOpenAIopenai_api_key = os.getenv("OPENAI_API_KEY", "YOUR_API_KEY")chat = ChatOpenAI(openai_api_key=openai_api_key, model='gpt-3.5-turbo', temperature=0.0, # Set 0.0 for consistent reply, easy to debug in dev)Intent Classification with GPT-3.5To understand whether a user is inquiring about books, movies, or other topics, we utilize GPT-3.5's advanced zero-shot learning capabilities for classification.class TopicClassifier: def __init__(self, llm): self.llm = llm self.topics = ["movies", "books", "others"] def classify(self, query): prompt = f"Classify the following question into one of these topics: '{','.join(self.topics)}': '{query}'" response = self.llm.predict(text=prompt, max_tokens=10) topic = response.strip().lower() return topicTailoring the Chatbot's Response with RAGAfter we know the user intention, we will trigger different process:Movies, LLM reply the user based on it knowldege and relvente info from our movie corpusBooks, LLM reply the user based on it knowldege and relvente info from our book corpusOthers, LLM reply the user based on it knowldegeWith the langChain agent, we can easy to decide this processfrom langchain.memory import ConversationBufferMemoryfrom langchain.agents import ConversationalChatAgent, AgentExecutorclass ChatAgent: def __init__(self, llm, tool_manager): self.llm = llm self.tool_manager = tool_manager self.memory = ConversationBufferMemory(memory_key="chat_history",input_key="input", return_messages=True) self.agent = ConversationalChatAgent.from_llm_and_tools(llm=self.llm, tools=list(self.tool_manager.tools.values()), system_message="You are a smart assistant whose main goal is to recommend amazing books and movies to users. Provide helpful, short and concise recommendations with a touch of fun!") self.chat_agent = AgentExecutor.from_agent_and_tools(agent=self.agent, tools=list(self.tool_manager.tools.values()), verbose=True, memory=self.memory) def get_response(self, query, topic_classifier): topic = topic_classifier.classify(query) tool_name = None if topic == "other" else topic.capitalize() + "Tool" try: response = self.chat_agent.run(input=query, tool_name=tool_name) if tool_name else self.llm.generate(prompt=query) except ValueError as e: response = str(e) return {"answer": response}Depending on what you ask, it either fetches the most relevant movie or book details from its database or crafts a direct response on its own. This flexibility ensures you get precise and engaging answers every time.SummaryBy combining the understanding capabilities of LLMs with the vast knowledge access provided by RAG, we've created a chatbot that's not just a tool but a companion ready to explore the world of books and movies with you.Hope you enjoy it and feel free to leave a comment.Machine LearningNLPGptLangchainRetrieval Augmented----1FollowWritten by Bill Huang140 FollowersFinding beauty in texts:)FollowHelpStatusAboutCareersBlogPrivacyTermsText to speechTeams

Chatbot with LLM and RAG in Action | by Bill Huang | Feb, 2024 | MediumOpen in appSign upSign inWriteSign upSign inChatbot with LLM and RAG in ActionBill Huang·Follow5 min read·Feb 28, 2024--1ListenShareIntroductionHello friends, in this article I will guild you through creating a cutting-edge chatbot for recommender with the power of LLM and with advanced retrieval technology to handle complex questions with ease.By the end, you'll learn how to:Build vector space for RAGBuild Chatbot power with LLM using LangChainMake chatbot's decison smarter with AgentLarge Language Models (LLMs)LLMs, such as GPT-4, Llama 2, Mixtral 8x7B, have revolutionized the way machines understand and interact with human language. Think of LLMs as the chatbot's brain, capable of deep understanding and generating human-like responses.Retrieval-Augmented Generation (RAG)RAG acts as a bridge, combining the creative response generation of models with the precision of retrieval-based methods, ensuring chatbots can source and integrate external information for more accurate responses.Traditional Chatbot Process vs. Chatbot + LLMTraditional chatbots often rely on scripted responses, limiting their flexibility. LLM integration expands their capabilities, enabling a dynamic and contextual conversation.Imagine traditional chatbots as librarians limited to their own knowledge, answering only from what they've learned.In contrast, LLM-enhanced chatbots are like librarians with access to every book in the whole library, providing wider-ranging and more detailed assistance.Chatbot vs. Chatbot + RAGChatbots without RAG are limited to responding based on their pre-existing knowledge, which can lead to incorrect or "hallucinated" answers when faced with unfamiliar questions. This means they might guess the answer, often resulting in unreliable responses.On the other hand, chatbots enhanced with RAG can access and use external information to answer questions. This capability allows them to provide more accurate and reliable responses, even for questions outside their initial training data. By integrating RAG, chatbots become more versatile and dependable for users seeking information.Imagine Chatbot + RAG are like librarians not only read all the book in the library, but in the middle of your conversation, can instantly fetch and read any relative info in everywhere to answer your questions.That's the power of RAG!Step-by-Step Code ExplanationNext, I will show you the proecss in buiding a book and movie recommender chatbot using Python, LangChain, embedding-based retrieval strategies, and OpenAI's GPT-3.5.The source code is here:GitHub - billpku/Chatbot_LLM_RAG_in_ActionContribute to billpku/Chatbot_LLM_RAG_in_Action development by creating an account on GitHub.github.comPreparation of the Corpus:The foundation of any RAG system is a robust and comprehensive corpus.For this chatbot, movie and book data were meticulously formatted into JSON, ensuring each entry had a unique ID, title, summary and etc. This standardized format facilitates efficient data retrieval and processing.My data were based on the the CMU's corpus:BookSummaryMovieSummaryBuilding the Vector Store for RetrievalTo enhance a chatbot's ability to search semantically similar content, we create embeddings for each item in the corpus. These embeddings are generated using sentence embedding models, such as the one from Hugging Face, and are indexed using FAISS for fast retrieval.Generating EmbeddingsFirst, we convert the content into embeddings using a sentence embedding model. This allows the chatbot to perform semantic similarity searches. Here's how to generate embeddings using the Hugging Face model:from langchain.embeddings import HuggingFaceEmbeddingsembeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")Storing and

Retrieving EmbeddingsNext, we need a service like FAISS to store these embeddings and retrieve the most similar ones based on user input. FAISS simplifies the storage and retrieval process for both book and movie recommendations. Here’s how to create, save, and load a FAISS store:from langchain.vectorstores import FAISS# Create FAISS store from documentvector_store = FAISS.from_documents(document, self.embeddings)# Save the spacevector_store.save_local(save_path)# Load the space, with embeddingsvector_store = FAISS.load_local(save_path, embeddings)Chatbot Logic and User Interaction:At its core, the chatbot is designed to interpret user queries, retrieve pertinent information, and generate responses that incorporate this information. This involves converting queries into embeddings, searching the corpus, and then using GPT-3.5 to craft a final response that integrates both the retrieved data and the model’s generative output.Using GPT-3.5 for Dynamic ResponsesWe employ GPT-3.5 to power our chatbot, enabling it to anticipate user needs and craft answers using our curated content.import osfrom langchain.chat_models import ChatOpenAIopenai_api_key = os.getenv("OPENAI_API_KEY", "YOUR_API_KEY")chat = ChatOpenAI(openai_api_key=openai_api_key,   model='gpt-3.5-turbo',   temperature=0.0, # Set 0.0 for consistent reply, easy to debug in dev)Intent Classification with GPT-3.5To understand whether a user is inquiring about books, movies, or other topics, we utilize GPT-3.5’s advanced zero-shot learning capabilities for classification.class TopicClassifier: def __init__(self, llm):   self.llm = llm   self.topics = ["movies", "books", "others"]   def classify(self, query):   prompt = f"Classify the following question into one of these topics: '{','.join(self.topics)}': '{query}'"   response = self.llm.predict(text=prompt, max_tokens=10)   topic = response.strip().lower()   return topicTailoring the Chatbot’s Response with RAGAfter we know the user intention, we will trigger different process:Movies, LLM reply the user based on it knowldege and relvente info from our movie corpusBooks, LLM reply the user based on it knowldege and relvente info from our book corpusOthers, LLM reply the user based on it knowldegeWith the langChain agent, we can easy to decide this processfrom langchain.memory import ConversationBufferMemoryfrom langchain.agents import ConversationalChatAgent, AgentExecutorclass ChatAgent: def __init__(self, llm, tool_manager):   self.llm = llm   self.tool_manager = tool_manager   self.memory = ConversationBufferMemory(memory_key="chat_history",input_key="input", return_messages=True)   self.agent = ConversationalChatAgent.from_llm_and_tools(llm=self.llm, tools=list(self.tool_manager.tools.values()), system_message="You are a smart assistant whose main goal is to recommend amazing books and movies to users. Provide helpful, short and concise recommendations with a touch of fun!")   self.chat_agent = AgentExecutor.from_agent_and_tools(agent=self.agent, tools=list(self.tool_manager.tools.values()), verbose=True, memory=self.memory)   def get_response(self, query, topic_classifier):   topic = topic_classifier.classify(query)   tool_name = None if topic == "other" else topic.capitalize() + "Tool"   try: response = self.chat_agent.run(input=query, tool_name=tool_name) if tool_name else self.llm.generate(prompt=query)   except ValueError as e: response = str(e)   return {"answer": response}Depending on what you ask, it either fetches the most relevant movie or book details from its database or crafts a direct response on its own. This flexibility ensures you get precise and engaging answers every time.SummaryBy combining the understanding capabilities of LLMs with the vast knowledge access provided by RAG, we’ve created a chatbot that’s not just a tool but a companion ready to explore the world of books and movies with you.Hope you enjoy it and feel free to leave a comment.Machine LearningNLPGptLangchainRetrieval Augmented----1FollowWritten by Bill Huang140 FollowersFinding beauty in texts:)FollowHelpStatusAboutCareersBlogPrivacyTermsText to speechTeams