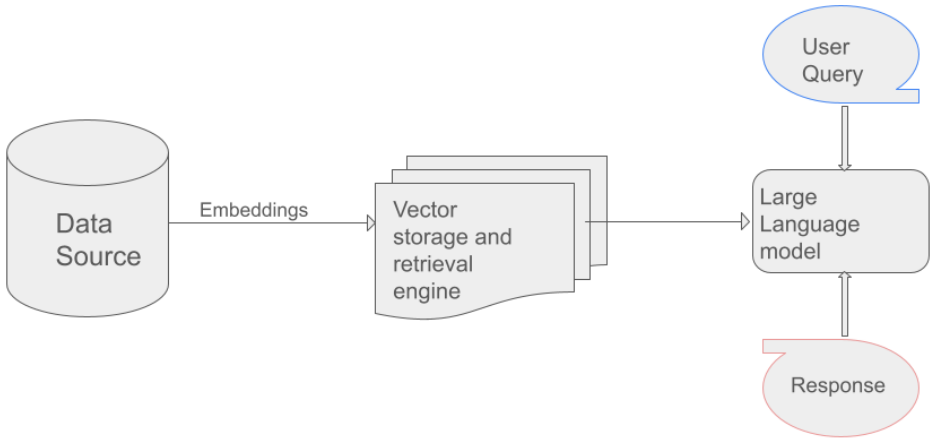
Technical Assessment task

I implemented a ChatGPT-like tool for information retrieval and streamlining access to custom documentation. Tailored to seamlessly provide answers from a document on one of my favorite novels, “The Nightingale” by Kristin Hannah, this tool is designed to interpret and respond to user queries in natural language.

Gone are the days of laborious searches through extensive documentation – this system combines the power of retrieval and generation techniques to offer concise and accurate responses derived exclusively from the selected documentation.

A Large language model confronts several challenges related to accuracy and knowledge cut-off. This is when an LLM returns information that is outdated because its knowledge is limited to the data available at the time of training.

Retrieval Augmented Generation (RAG) offers a technique to mitigate these challenges by providing the model access to external data sources. I used Retrieval Augmented Generation which works with pretrained LLMs and our data to generate responses.



Retrieval Augmented generation

Figure 1. Architecture diagram

How is the data from the document stored?

Data ingestion is a crucial part of many natural language processing (NLP) systems where textual data needs to be prepared and transformed into a format suitable for downstream tasks, such as training machine learning models or answering user queries.

- Data Source: I retrieved the data from the synopsis of one of my favorite novels by Kristin Hannah, “The Nightingale” as available on Wikipedia.
- To ensure that the performance of the Retrieval Augmented Generation was good, I cleaned this data by taking the following steps:
    - Redundancies, such as repetitive or duplicate entries, were systematically removed to improve the overall efficiency of the system.
    - Topics were categorized based on key aspects, including the overarching theme, impact, relevance, and the portrayal of characters within the novel.

```
loader = UnstructuredFileLoader("The_Nightingale.txt")
raw_documents = loader.load()
print("Document loaded using document_loaders from LangChain")
```

Figure 2. Document loaders from LangChain

- Once the data is loaded, it needs to be broken down into smaller, manageable pieces or "chunks." Language models, huge ones like GPT (Generative Pre-trained Transformer), often have limitations on the amount of text they can process at once. Breaking down text into chunks allows for more efficient processing. The "chunk size" is a parameter that needs to be tuned based on the specifics of the language model and the requirements of the task.
- I experimented with the chunking approach. I looped through each set once with a small, medium, and large chunk size and found small (500) to be best.

```
text_splitter = CharacterTextSplitter(
    separator="\n\n",
    chunk_size=500,
    chunk_overlap=100,
    length_function=len,
)
documents = text_splitter.split_documents(raw_documents)
print("Text splitted using text_splitter from LangChain")
```

Figure 3. Text Splitters from LangChain

- For each chunk of text, a numerical embedding is created. An embedding is a vector representation of the text that captures its semantic meaning in a numerical form.

- Embeddings are important for downstream tasks, as they allow the system to compare and measure the similarity between different pieces of text. Similarity in the embedding space is used to identify relevant chunks when answering questions or performing other tasks.
- The embeddings generated in the previous step are loaded into a "vectorstore." A vectorstore is a data structure that facilitates efficient storage and retrieval of vector embeddings.

```
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(documents, embeddings)
print("created a vectorestore using FAISS from LangChain")

vectorstore.save_local("faiss_index_constitution")
print("Vectorestore saved in 'faiss_index_constitution' ")
```

Figure 4. Creating vectorstore using FAISS from LangChain

How is the data queried?

- **Questions from the user:** To retrieve the questions from the user, I created a command-line application and a Web Interface to enhance the user's experience.
- **Chat history:** The incoming user's questions are merged with the chat history. The chat history is essential for maintaining context and allowing users to ask follow-up questions. By combining the new question with the chat history, the system creates a unified, standalone question that incorporates the context of the ongoing conversation.
- **Leveraging the vectorestor:** The system performs a lookup to identify relevant documents for the given question. The vectorstore efficiently stores the numerical embeddings of chunks of text, allowing for quick similarity searches. The goal is to retrieve documents or text chunks that are semantically similar or contextually relevant to the user's question. This step helps narrow down the information space to focus on potentially useful content.
- **Response generation:** A language model, often a powerful pre-trained model like GPT, is capable of understanding context, syntax, and semantics to formulate coherent and contextually relevant responses. Hence, this system employs a language model to generate a response.

```
def qa_chain():
    llm = ChatOpenAI(model_name="gpt-4", temperature=0.5)
    retriever = load_retriever()
    memory = ConversationBufferMemory(
        memory_key="chat_history", return_messages=True)
    model = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=retriever,
        memory=memory,
        combine_docs_chain_kwargs={"prompt": QA_PROMPT})
    return model
```

Figure 5. Question Answering system using ChatOpenAI

- Here, I experimented with different ways to define a question-answering (QA) system using a combination of language models, a retriever for document lookup, and a conversation buffer memory to store chat history
- The chosen model is "gpt-4," and the temperature parameter (0.1) influences the randomness of the model's responses during generation.
- The memory key is set to "chat_history," and return_messages=True indicates that the memory should include individual messages for later retrieval or analysis.
- This function uses the ConversationalRetrievalChain.from_llm method to create a conversational retrieval chain model.
- The ‘combine_docs_chain_kwargs’ parameter includes additional keyword arguments for configuring the behavior of the retrieval chain, such as the prompt used for combining documents.

```
template = """You are an AI assistant for answering questions about the story of the novel The Nightingale by author Kristin Hannah.
Ensure responses are specific to the novel and presented in a clear, concise manner.
Question: {question}
=========
{context}
=========
Answer in Markdown:"""
```

Figure 6. Prompt for Question Answering system

- Nevertheless, when prompts were supplied, the model exhibited inferior performance compared to its performance without prompts, resulting in an average response time of approximately 20 seconds instead of 8 seconds.

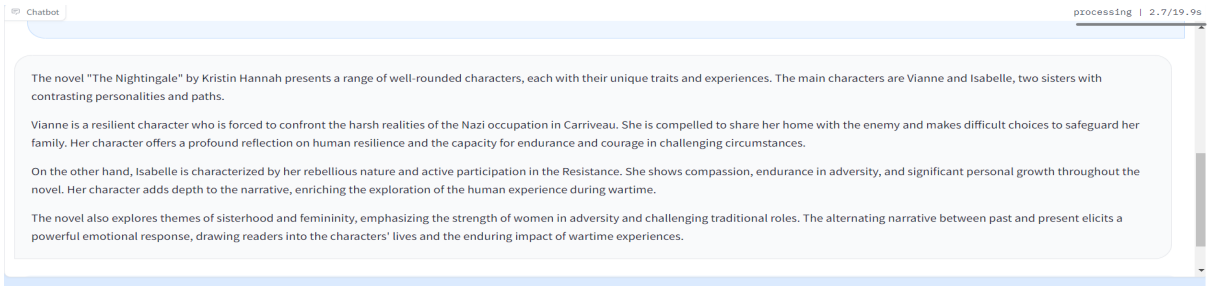


Figure 7. High average response time using the prompt for the Question Answering system

How is the model evaluated?

- The Rouge scores are metrics used to evaluate the quality of automatic summarization or text generation models. It is based on the count of overlapping n-grams between the system output and the reference summaries.
- For example: Rouge-1 (Rouge-N with N=1): Measures the overlap of unigram (single-word) sequences between the generated text and the reference text.
- I have created the following:
    - prediction list: A list of predictions to score. Each prediction is a string with tokens separated by spaces.

    - Reference list: A list of references for each prediction or a list of several references per prediction. Each reference is a string with tokens separated by spaces.

{'rouge1': 0.4162257495590829, 'rouge2': 0.23522224138922432, 'rougeL': 0.3280423280423281, 'rougeLsum': 0.3650793650793651}

Ultimately, some amount of human evaluation will be required to assess the results out of the knowledge source.

How to set up the tool?

- Run the following in the cmd prompt, replacing <yourkey> with your API key:
    - For Windows: setx OPENAI_API_KEY “<yourkey>”
- Inside the folder for the code, install the required libraries using this command:
    - pip install -r requirements.txt
- Run the ‘data_storage.py’ file using:
    - python ingest_data.py
- Interact with the tool by running the ‘chat_bot_cli.py’ for the command-line application or the ‘chat_bot_ui.py’ file for the Graphical User Interface (GUI):
    - python ‘chat_bot_cli.py
    - python ‘chat_bot_ui.py

Note: Ensure that you have the Python 3.11 interpreter installed on your system for proper execution.