

RAG Chatbot

Rian Radeck Santos Costa

25 de Outubro de 2023

1 Introdução

Para implementar um RAG chatbot foi necessária uma pesquisa para encontrar fontes qualificadas a fim de auxiliar durante o decorrer do trabalho. Cito, como exemplos, o seguinte:

- Documentação da OpenAI
- Tutoriais do Streamlit para deploy de um chatbot
- Documentação do Chroma DB
- Documentação do LangChain
- Discussões no Github do LangChain

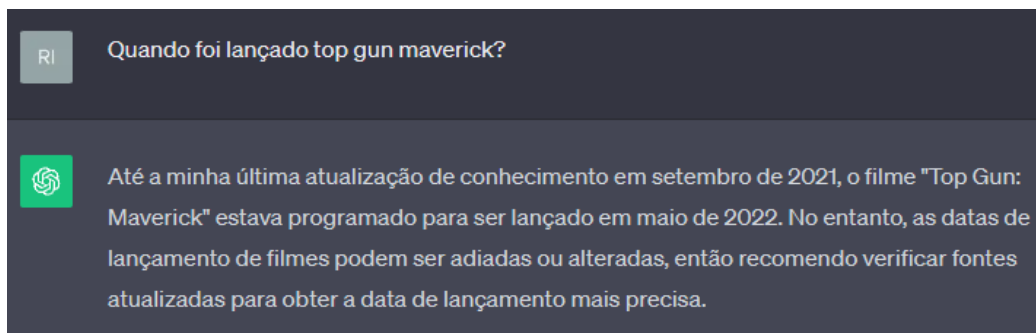
A leitura prévia dos tópicos acima mencionados contribuiu de forma significativa para o desenvolvimento do proto, reforçando e complementando meu conhecimento prévio sobre LLMs.

2 Funcionamento

Nessa seção vamos explorar o funcionamento de um RAG chatbot.

2.1 Motivação/Problema

O problema que procuramos resolver é suprir a falta de informação em modelos que já foram treinados. Um bom exemplo é o ChatGPT, que em seu treinamento usou dados até agosto de 2021, o que o torna alheio sobre eventos e notícias recentes. No caso específico do ChatGPT esse problema é mitigado com o uso de reinforcement learning a partir de dados fornecidos pelos seus usuários. Podemos ver esse comportamento na imagem a seguir



2.2 Solução

Para resolver esse problema devemos programar um chatbot que consiga fazer uma busca de contexto em uma base de dados de acordo com o prompt do usuário e envie tanto o prompt quanto o contexto para nosso LLM.

De maneira bem objetiva, o passo a passo pode ser descrito da seguinte maneira:

1. Receber a entrada do usuário.
2. Buscar nos dados locais/privados seções de texto que tenham grande similaridade com a entrada, ou seja, que tenham o mesmo contexto.
3. Enviar todo o contexto necessário juntamente com a entrada do usuário para o modelo. Dessa forma ele possuirá informação suficiente para responder à pergunta.

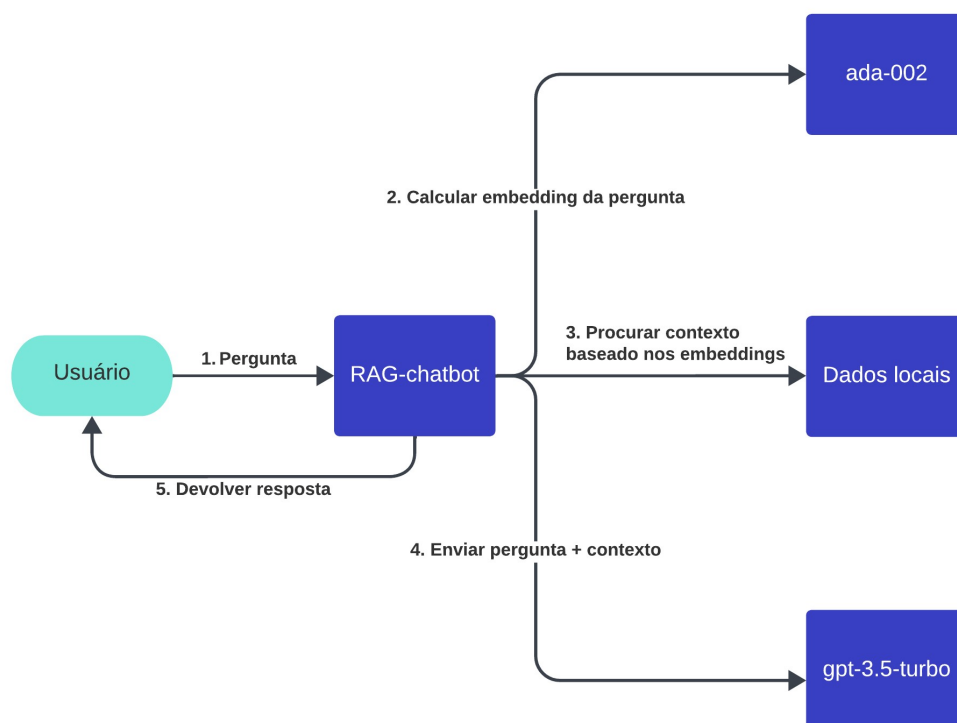
Sendo assim, nosso grande desafio é justamente definir quais partes do texto possuem o mesmo contexto da pergunta.

A ideia para resolver nosso desafio é simples: representar o texto de maneira numérica (e esperta) para que textos com “números” próximos possuam maior similaridade, ou seja, sejam V_1 e V_2 duas representações numéricas de textos diferentes, a sua similaridade será definida por $1/\|V_1 - V_2\|$ (essa função é apenas para termos um registro pedagógico). Sendo assim, quanto menor a diferença, maior a similaridade.

A maneira pela qual representaremos nossos textos se baseia em **embeddings**, ou seja, um texto que é dividido em tokens passará por uma função numérica (em nosso caso um modelo da OpenAI, ada-002) que nos retornará um vetor de pontos flutuantes.

Finalmente para descobrirmos quão similar são dois textos basta aplicarmos uma função chamada similaridade dos cossenos, que recebe como parâmetros dois vetores, que são os embeddings desses textos.

Sendo assim um esquema completo de uma requisição pode ser bem resumida no esquema abaixo:



3 Implementação

Agora que sabemos todo o funcionamento de um RAG chatbot, vamos explorar os detalhes da implementação.

3.1 Tratamento de dados

A parte que considero mais importante é preparar os dados para que a busca de contexto seja efetiva. Para atingir essa eficiência devemos dividir

nossos dados em grandes baldes (chunks) de texto. Na busca, escolheremos os melhores chunks, independente de estarem próximos no texto original. Em nosso caso específico utilizamos o texto da página HTML sobre o vestibular da Unicamp e dividimos ele em chunks de 1000 caracteres. Observe que o tamanho dos chunks e seu overlap são hiper parâmetros de nosso chatbot e podem ser ajustados com um conjunto de validação.

Outro detalhe importante é a interpretação de tabelas da página, onde foi feito tratamento manual com a ajuda de um parser de tabelas HTML, pois apenas o texto HTML não fornecia informações sobre o significado de uma célula na tabela. O tratamento de tabelas nos permite responder perguntas como: “Quantas vagas serão disponibilizadas ao curso de ciência da computação pelo vestibular Unicamp 2024?” e “Qual a área do curso de enfermagem segundo o vestibular Unicamp 2024?”.

O refinamento feito sobre as tabelas pode ser facilmente estendido para outros aspectos do texto, como os artigos do edital, datas e documentos importantes, por exemplo.

Como ponto importante, também cito a necessidade de se identificar qual separador usamos para dividir nossos chunks, que em nosso caso foi o ponto final e a quebra de linha, permitindo continuidade de contexto dentro do próprio chunk, ou seja, sem quebrar a linha de pensamento abruptamente.

3.2 Construtor da classe

Agora que nosso dado está bem tratado, com a explicação acima, vamos para a parte mais legal: o código! Nosso primeiro passo é definir os parâmetros do chatbot. Vamos olhar como isso foi feito:

```
1  def __init__(self):
2      """
3          The constructor of the chatbot
4
5          It is responsible for reading and splitting the data,
6          and also to calculate the embeddings
7          for the vector database.
8
9          We are using the Chroma DB as a vector database to
10         use as a retriever for our RAG.
11         """
12
13         # Load the data.
14         self.loader = TextLoader("out.txt", encoding='utf-8')
```

```

13         self.documents = self.loader.load()
14
15         # Split the data into a thousand-ish characters
16         chunks.
17         self.text_splitter = RecursiveCharacterTextSplitter(
18             chunk_size=1000, chunk_overlap=0, separators=[".", "\n"])
19         self.docs = self.text_splitter.split_documents(self.
20             documents)
21
22         # Define the openai embedding function to be used in
23         the vector database.
24         self.embedding_function = OpenAIEmbeddings()
25
26         # Initialize the vector db with the information about
27         the data and the embedding function.
28         self.db = Chroma.from_documents(self.docs, self.
29             embedding_function)
30
31         # Last but not least, initialize the chat history as
32         an empty list.
33         self.chat_history = []

```

No construtor acima, declaramos todos os componentes do nosso chatbot: dados, função de embedding e o histórico de conversação.

O arquivo out.txt é o texto que será dividido e o histórico do chat é utilizado para manter contexto sobre a própria conversa, já que a maneira que decidi abordar o problema foi enviar também o contexto histórico para o gpt-3.5-turbo.

3.3 Função principal

Com os componentes definidos e inicializados, vamos discutir a implementação da requisição. Apesar de entender a ideia geral de funcionamento, na implementação precisamos tomar cuidado com um importante detalhe: a quantidade de tokens da nossa requisição, pois o modelo escolhido (gpt-3.5-turbo) aceita apenas 4096 tokens.

Outro ponto de atenção, como dito no final do tópico anterior, é que devemos enviar também o histórico de conversa, pois não queremos que o chatbot esqueça o contexto histórico da conversa e isso nos traz um dilema: como balancear o contexto histórico do chat com o contexto do dado?

Vamos explorar a maneira como eu abordei esse dilema: Escolhi que em toda requisição será necessário ao menos 1 chunk de contexto de dados, pois

assim a integridade do chatbot é mantida. Após calcularmos a quantidade de tokens do melhor contexto mais a quantidade de tokens da pergunta, vamos colocar o máximo de contexto histórico possível, ou seja, adicionar o histórico do chat enquanto houverem tokens disponíveis. Por último, se sobraem tokens devemos adicionar mais contexto do nosso dado, pois quanto mais, melhor.

De maneira resumida:

1. Adicionar pergunta e melhor contexto de dados à requisição.
2. Enquanto houver espaço e contexto histórico, adicionar contexto histórico.
3. Enquanto houver espaço e contexto de dados, adicionar contexto de dados.

Os passos da implementação podem ser observados no código a seguir:

```
1  def get_response(self, query):
2      """
3          Gets the response of a query with the OpenAI API
4          sending the query,
5          the chat history, and the most relevant chunks of
6          context from the vector DB.
7      """
8
9      # Get the most relevant chunks from the vector db
10     given the query.
11     docs = self.db.similarity_search(query)
12
13     # Initialize the message that will be sent to the
14     OpenAI API.
15     # We will send no more than 4 thousands tokens for
16     the API.
17     messages = [{"role": "user", "content": query}]
18
19     # Calculates the number of tokens of the message that
20     will be sent (currently)
21     # that consists of the query and the most relevant
22     chunk for context.
23     num_tokens = len(get_tokens(query)) + len(get_tokens(
24     docs[0].page_content))
25
26     # Put as much chat history as possible keeping track
27     of the total number of tokens in the message.
28     i = 1
```

```

20         while i-1 < len(self.chat_history) and num_tokens +
21             len(get_tokens(self.chat_history[-i]["content"])) <= 4000:
22                 num_tokens += len(get_tokens(self.chat_history[-i]
23                     ["content"]))
24                 messages.append(self.chat_history[-i])
25                 print(f"I've put {self.chat_history[-i]}")
26                 i += 1
27                 messages.append({"role" : "system", "content": docs
28                     [0].page_content})
29
30             # Put as much context as possible keeping track of
31             the total number of tokens in the message.
32             i = 1
33             while i < len(docs) and num_tokens + len(get_tokens(
34                 docs[i].page_content)) <= 4000:
35                 num_tokens += len(get_tokens(docs[i].page_content
36                     ))
37                 messages.append({"role" : "system", "content":
38                     docs[i].page_content})
39                 i += 1
40
41             # Put the message in the correct message for the
42             OpenAI API.
43             messages = messages[::-1]
44
45             # Sends the message
46             completion = openai.ChatCompletion.create(
47                 model="gpt-3.5-turbo",
48                 messages=messages
49             )
50
51             # Add the query and the response to the chat history
52             self.chat_history.append({"role" : "user", "content":
53                 query})
54             self.chat_history.append({"role" : "assistant", "
55                 content": str(completion.choices[0].message.content)})
56
57             # Returns the response
58             return completion.choices[0].message

```

Por último, implementei o deploy do chatbot com o Streamlit, onde tomei como inspiração o Echobot na página de tutorial da biblioteca. Fazendo apenas algumas modificações em seu código podemos adicionar uma instância do nosso chatbot em nosso estado da sessão do streamlit e mudar a resposta para nossa função `get_response`.


```

1 # based on https://docs.streamlit.io/knowledge-base/tutorials
  /build-conversational-apps
2
3 import streamlit as st
4 from main import RAG
5
6
7 st.title("RAG Bot")
8
9 # Initialize chat history
10 if "messages" not in st.session_state:
11     st.session_state.messages = []
12 if "bot" not in st.session_state:
13     st.session_state.bot = RAG()
14
15 # Display chat messages from history on app rerun
16 for message in st.session_state.messages:
17     with st.chat_message(message["role"]):
18         st.markdown(message["content"])
19
20 # React to user input
21 if prompt := st.chat_input("Pergunte algo sobre o vestibular
  Unicamp 2024"):
22     # Display user message in chat message container
23     st.chat_message("user").markdown(prompt)
24     # Add user message to chat history
25     st.session_state.messages.append({"role": "user", "
  content": prompt})
26
27     response = str(st.session_state.bot.get_response(prompt).
  content)
28     # Display assistant response in chat message container
29     with st.chat_message("assistant"):
30         st.markdown(response)
31     # Add assistant response to chat history
32     st.session_state.messages.append({"role": "assistant", "
  content": response})

```

4 Performance e possíveis melhorias

Para testar a performance do chatbot, fiz perguntas específicas sobre o edital, algumas mais fáceis e outras mais difíceis. Fiquei satisfeito com o resultado, mesmo consciente que o desempenho pode ser aperfeiçoado.

Vamos para as perguntas e respostas. Perguntas com * são consideradas difíceis:

- Usuário: Qual a área do curso de enfermagem segundo o vestibular Unicamp 2024?
- Chatbot: De acordo com o Manual de Orientações para o Vestibular Unicamp 2024, o curso de Enfermagem da Unicamp está inserido na área de Ciências da Saúde.
- Usuário*: Quantas vagas serão disponibilizadas ao curso de ciência da computação pelo vestibular Unicamp 2024?
- Chatbot: No Vestibular Unicamp 2024, serão disponibilizadas 40 vagas para o curso de Ciência da Computação.
- Usuário: Qual a quantidade de vagas total no vestibular Unicamp 2024?
- Chatbot: No Vestibular Unicamp 2024, serão oferecidas um total de 3.340 vagas regulares no ensino de graduação da Unicamp.
- Usuário*: Qual a data das provas do vestibular?
- Chatbot: A primeira fase do Vestibular Unicamp 2024 será realizada no dia 29 de outubro de 2023. Já a segunda fase ocorrerá nos dias 3 e 4 de dezembro de 2023. É importante ressaltar que essas datas podem estar sujeitas a alterações, portanto, é recomendado verificar o calendário oficial da Comvest para obter informações atualizadas.

Assim, percebe-se que o chatbot de fato consegue utilizar o contexto do edital para nos trazer informações específicas sobre o vestibular Unicamp 2024, embora ainda possa melhorar, e muito.

Para aperfeiçoarmos o chatbot, primeiro precisaremos definir outros critérios para uma avaliação métrica objetiva, utilizando um dataset de perguntas e respostas correspondentes, seguindo-se de uma análise do quanto há de acertos e eventuais erros. A automatização do processo poderia fazer o uso

do próprio ChatGPT para comparar a resposta do chatbot com a resposta esperada e nos dizer se elas significam a mesma coisa.

Dado uma métrica de desempenho, os principais pontos de melhoria na minha opinião seriam:

- Melhor tratamento dos dados.
- Tunagem de hiper parâmetros do chatbot.
- Uso de um LLM mais poderoso (como o GPT-4, por exemplo).
- Outros dados sobre o vestibular, explicados em uma linguagem mais simples e amigável (os modelos gostam disso).

5 Conclusão

Vale registrar que esse documento serve de auxílio para entender o projeto com o código completo, que está localizado no meu Github.

Após todas as etapas descritas acima, conseguimos entender o funcionamento e a implementação de um chatbot baseado em RAG. Vimos também como o uso de bibliotecas pode nos auxiliar nessa tarefa e como fazer um deploy simples e rápido de nosso chatbot.

Reforçando a leitura dos links fornecidos no início do documento e do código fonte, espero que esse relatório possa auxiliar o leitor a ter uma boa experiência para o problema posto como desafio.

P.S.: Esse documento descreve o estado do robô no dia 25 de outubro de 2023, é possível que mudanças e aprimoramentos tenham sido feitos já que estou gostando bastante de implementar essa aplicação.