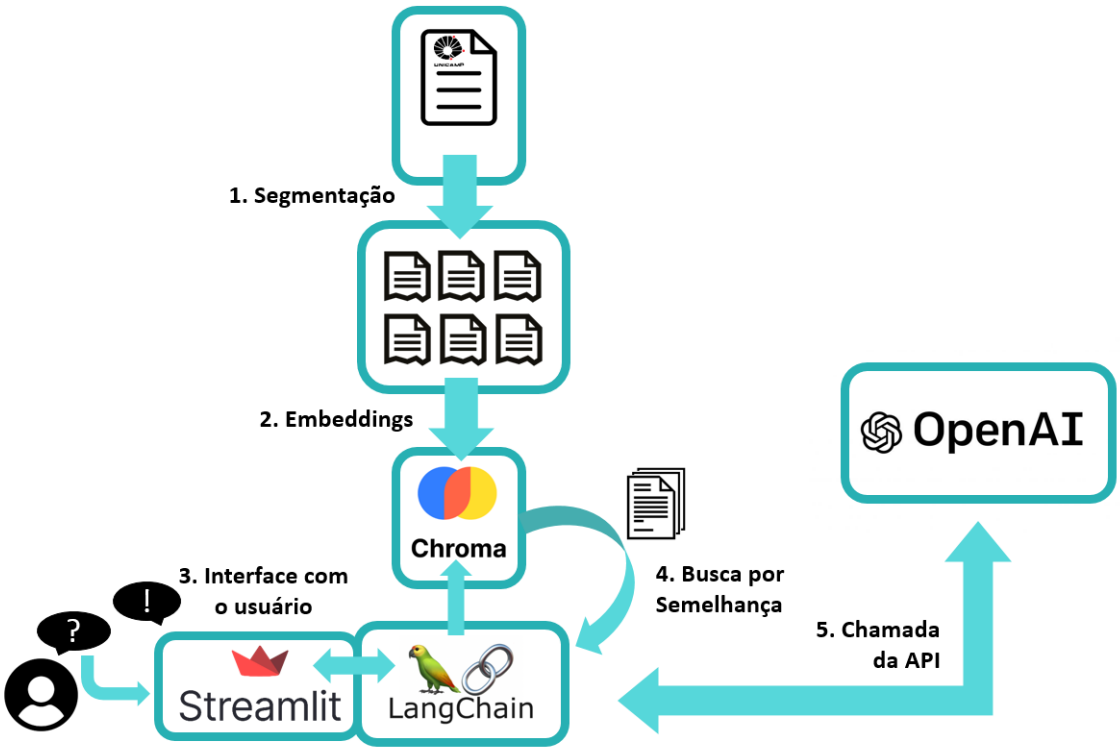
Projeto: Chatbot para Responder Dúvidas sobre o Vestibular da Unicamp 2024

Ana Carolina Tavares Sovat

Índice

Visão Geral do Projeto



O intuito inicial do projeto era desenvolver um chatbot capaz de responder perguntas sobre o Vestibular Unicamp 2024, baseando-se no documento da Resolução GR-031/2023, e utilizando a API do ChatGPT (OpenAI). O chatbot desenvolvido possui uma arquitetura básica de *Retrieval Augmented Generation*, utilizando o *framework* LangChain para acessar a API da OpenAI e buscar informações relevantes no documento. Foram utilizadas também as bibliotecas Chroma e Streamlit, para fazer a armazenagem dos vetores de embedding e a interface/hospedagem do chatbot, respectivamente.

O funcionamento de um sistema de RAG pode ser resumido em uma sequência simples de passos. Primeiramente, a pergunta feita pelo usuário ao chatbot é transformada em uma representação vetorial (*embedding*), e comparada com um banco de dados de pequenos fragmentos de texto (também em formato de *embedding*), buscando encontrar possíveis fragmentos relevantes para a resposta da pergunta, baseando-se na distância dos vetores. Neste chatbot, essa etapa é realizada pelo sistema de armazenagem de vetores (*vectorstore*) Chroma, manipulado pelo *framework* LangChain. De posse dos fragmentos mais relevantes para a resposta, o programa então faz uma requisição ao modelo de linguagem, baseado em *prompts*, incluindo os fragmentos relevantes como contexto para a pergunta. Aqui, o modelo escolhido foi o ChatGPT, acessado através de sua API, também integrada ao LangChain. A resposta do modelo é então processada e devolvida ao usuário.

A construção do chabot pode ser detalhada, portanto, em cinco etapas distintas:

1. Segmentação do Documento Base
2. Geração dos Embeddings

3. Desenvolvimento da Interface + Hospedagem do Chatbot na Nuvem
4. Busca por Semelhança na Base de Dados
5. Chamada da API da OpenAI

Segmentação

A etapa de segmentação e pré-processamento do documento base foi essencial para este projeto. Como se tratava de um único documento, relativamente pequeno (em uma escala de bases de conhecimento) e estruturado, optei por fazer a segmentação do texto respeitando as quebras lógicas do documento. Dessa forma, os trechos recuperados como relevantes para a resposta são relativamente completos em si mesmos, com sentenças completas e idéias que podem ser entendidas sem o resto do documento, na maioria dos casos. Isso traz vantagens para o sistema final, pois assume-se que o modelo consegue mais informações com os trechos completos.

Da mesma forma, quando possível, adicionei os nomes das seções ou expressões chaves ao início dos trechos que precisavam de um pouco mais de contexto, descartando esses mesmos nomes de seções dos documentos finais. Essa estratégia busca utilizar a estrutura do documento a favor do sistema de busca, evitando situações em que, por exemplo, um trecho que contenha o título da seção (ex. “Conteúdo das provas”) seja considerado como muito mais relevante para o tema do que os seguintes, apesar de todos estarem na mesma seção, e serem igualmente pertinentes a uma pergunta sobre o assunto. Dessa forma, busquei preservar a hierarquia das informações, para gerar resultados mais relevantes na busca por fragmentos.

O tamanho adequado dos fragmentos também foi objeto de investigação e experimento. Num primeiro momento, me atentei ao tamanho máximo para as *embeddings* anunciado pelo modelo, de 8191 tokens. Após fazer a segmentação, utilizei a biblioteca Python TikToken para calcular o tamanho, em tokens, de cada segmento, e garantir que não estavam muito grandes. Ainda assim, encontrei posteriormente outro gargalo para o tamanho dos fragmentos, ao fazer a chamada para a API da OpenAI. Como meu sistema seleciona os 4 trechos mais semelhantes à pergunta, e os insere no *prompt* enviado para a API, o tamanho máximo (em tokens) para esse *prompt* logo se tornou um problema. Sendo assim, refiz a segmentação do documento, levando em consideração o tamanho médio dos fragmentos (cuja combinação vai ser determinada pelo sistema de busca) para que não somassem, juntos, mais de 4.000 tokens. Ao mesmo tempo, busquei não dividir seções que precisariam ser encontradas em sua totalidade, a depender da pergunta, em mais de 4 partes. Ou seja, foi um *trade-off* a ser experimentado, mas, de forma geral, descobri que fragmentos menores, com informações mais específicas, melhoram a acurácia da busca por fragmentos.

Finalmente, algumas seções do documento continham informações importantes descritas em tabelas. Como o modelo utilizado é voltado para o entendimento de texto, essas informações precisavam ser entregues para o modelo de outra forma. Em uma ocasião (Tabela “Relação de Livros”), optei por traduzir a tabela em texto corrido, e inseri esse trecho nos fragmentos da base de dados.

Embeddings

Para gerar a base de dados do chatbot, os fragmentos já pré-processados foram vetorizados, utilizando as próprias *embeddings* da OpenAI. As representações vetoriais são então armazenadas na base de dados Chroma, que faz as operações de comparação e busca nesses vetores de forma eficiente.

Em próximos trabalhos, pode-se avaliar o desempenho de outras *embeddings* para esta tarefa. Quanto melhores as *embeddings*, melhor será a representação vetorial dos textos, e melhores serão os resultados da busca por semelhança. As *embeddings* podem ser trocadas sem prejuízo de compatibilidade com o resto do modelo, pois são utilizadas apenas na etapa de busca dos fragmentos de texto relevantes, não se comunicando diretamente com o modelo de linguagem que interpretará o *prompt* gerado.

Foi feita alguma experimentação com diferentes funções de cálculo de distância de vetores (produto interno, semelhança de cosseno, “L2 quadrado”), mas concluiu-se afinal que a função original (“L2 quadrado”) apresentava os melhores resultados.

Interface

Este chatbot utiliza a *framework* Streamlit para gerar sua interface visual. Essa biblioteca Python torna extremamente fácil o desenvolvimento de uma página web interativa, a partir de qualquer programa em Python, e também se encarrega de servir a página em uma porta da máquina em que roda. O *framework* também conta com um serviço de hospedagem na nuvem gratuito (baseado na nuvem do GitHub), o que permite ao desenvolvedor trabalhar de forma remota, utilizando a IDE CodeSpaces do GitHub, e configurando seu ambiente de forma fácil, sem restrições de hardware locais.

Contando com todas essas facilidades, mais os widgets já prontos para a geração de chatbots, foi possível desenvolver um chatbot funcional, com uma interface apresentável, em pouco tempo, focando apenas no desenvolvimento do sistema em si.

Busca

A *framework* LangChain é que conecta todo o fluxo de dados da execução do chatbot. Ao receber da interface a pergunta do usuário, ela consulta a base de dados, buscando os fragmentos relevantes, e então concatena a pergunta e os fragmentos selecionados em um *prompt*, que envia à API da OpenAI, para enfim retornar a resposta da API para a interface.

Inicialmente, cogitou-se usar diretamente a biblioteca da OpenAI para desenvolver o projeto, mas depois de um estudo rápido das capacidades de integração da biblioteca LangChain, optou-se por esta ferramenta, que reúne as capacidades de outras tantas bibliotecas, além da OpenAI e Chroma, utilizadas no projeto.

Foram feitos alguns experimentos com o parâmetro do número de fragmentos a serem retornados pelo buscador, mas por fim, decidiu-se por manter o padrão de 4 trechos retornados, buscando um equilíbrio entre aumentar a quantidade de material retornado, a fim de produzir respostas mais informadas, e evitar atingir o tamanho máximo do *prompt* a ser enviado para a API, como descrito anteriormente. A definição desse parâmetro foi feita,

portanto, em conjunto com a definição do tamanho médio dos trechos do documento. Percebeu-se que com mais fragmentos retornados, maior a chance de serem combinados vários trechos grandes, e atingir-se o limite do *prompt*, mas 4 trechos pareceu ser o mínimo, dada a precisão da busca, para que os fragmentos relevantes tivessem mais chance de serem encontrados.

Chamada da API

Finalmente, o desenvolvimento do *prompt* para interagir com a API da OpenAI foi também importante para influenciar o comportamento e o tom da resposta do modelo. As instruções tentam evitar que o modelo tente responder às perguntas com seu próprio conhecimento, o que abre brecha para a ocorrência de alucinações. Já que o grande ganho de um chatbot baseado em RAG sobre aqueles que utilizam apenas um modelo de linguagem para responder às perguntas, é justamente a maior confiabilidade de suas respostas, busquei instruir o modelo a se ater às informações fornecidas pelo buscador de trechos do documento, respondendo que não sabe a resposta, quando essas informações não estiverem disponíveis. Além disso, busquei modificar o tom das respostas, que inicialmente me pareceram demasiadamente formais, secas e verbosas.

Projetos Futuros

Muitas *features* imaginadas para este projeto tiveram que ser abandonadas por não poderem ser implementadas a tempo. A seguir, descreverei um pouco o que havia imaginado.

Primeiramente, gostaria de ter implementado um sistema de logs, para armazenar todas as perguntas feitas pelos usuários, juntamente com as respostas recebidas, a fim de monitorar e registar o desempenho do chatbot, enquanto fazia testes. Durante o processo de desenvolvimento, pude disponibilizar o chatbot (utilizando o serviço de hospedagem do Streamlit) para amigos e familiares, que me ajudaram a perceber falhas iniciais. Neste processo, um sistema de logs me permitiria monitorar esse desempenho de maneira mais sistemática.

Uma *feature* que é sempre interessante de ser implementada em chatbots é a manutenção do contexto da conversa, que está ausente neste projeto. Acredito que não seria muito complexo acrescentar essa feature, mas ainda seria necessário estudar um pouco mais o funcionamento do Streamlit, para ver o que é guardado e o que é sobrescrito a cada interação do usuário com a página. Se for possível guardar o último input do usuário, localmente, persistindo na nova execução do código, será possível combinar o input antigo com o novo, e enviá-los juntos na nova chamada da API, o que é simples de ser feito através do prompt.

Buscando capitalizar em cima da maior confiabilidade das respostas de um chatbot baseado em RAG, que é seu grande diferencial, gostaria de retornar para o usuário, juntamente com cada resposta, em que artigos ou trechos ela foi baseada. Acredito que isso não seria muito difícil de ser feito, bastando associar, a cada trecho do documento original, o texto completo do artigo de que ele foi retirado, em uma tabela. Como internamente, o modelo retorna os trechos consultados na elaboração da resposta, seria possível recuperar o texto

completo do artigo mais relevante, por exemplo. Também existe a possibilidade de trabalhar com estruturas hierárquicas de trechos disponíveis no *framework* LangChain (ParentDocumentRetriever), que facilitariam esse processo. Haveria, ainda, o trabalho de desenvolver uma maneira de apresentar essa informação para o usuário, talvez na forma de um link para o texto completo do artigo, em outra página, ou em outra estrutura, como uma barra lateral ou uma janela pop-up.

A *feature* mais interessante que não pude implementar, no entanto, foi a chamada de funções pelo modelo de linguagem. Acredito que essa seria a solução ótima para o problema da leitura de dados de tabelas, que podem ser facilmente acessadas, de forma simples, por funções. Com as ferramentas da *framework* LangChain, é possível disponibilizar funções arbitrárias para o modelo de linguagem, que decide quando chamá-las, de acordo com a pergunta feita pelo usuário, baseando-se na documentação presente no código da função. Para acrescentar essa feature no modelo, bastaria apenas fazer a integração dessa funcionalidade com o sistema RAG já implementado.

Conclusão e Aprendizados

Com este projeto, aprendi o básico da utilização dos *frameworks* LangChain e Streamlit, com os quais não havia tido contato anteriormente. Ambos se revelaram bastante úteis, poderosos, e práticos para desenvolvimento em pouco tempo.

Também tive a experiência de implementar um sistema RAG na prática, entrando em contato com os desafios e tecnologias dessa abordagem.

Acredito que o resultado final do projeto ficou surpreendentemente bom, para tão pouco tempo de desenvolvimento. A partir dos testes preliminares que realizei, considero que o desempenho do chatbot está bastante razoável, fornecendo respostas corretas para as principais perguntas que imagino que devem ser feitas para este sistema. Em meus testes, perguntei sobre datas e conteúdo das provas, documentação necessária, taxa de inscrição, acomodações para deficientes, entre outros temas. Ao receber uma resposta não adequada, busquei descobrir a origem do problema, e resolvê-lo, seja alterando parâmetros, ou melhorando o pré-processamento dos dados.

Uma bateria de testes mais sistemáticos (ainda com resultados avaliados de forma manual) seria vantajosa para se ter uma visão geral das capacidades do chatbot, na forma presente, mas infelizmente não pôde ser realizada por restrições de tempo.