UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Customização de modelos de linguagem por meio de bancos de dados vetoriais

Murilo Mussatto

São Carlos – SP

Customização de modelos de linguagem por meio de bancos de dados vetoriais

Murilo Mussatto

Orientador: **Prof. Dr. Jose Fernando Rodrigues Júnior**

Monografia final de conclusão de curso do Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação – ICMC-USP, para obtenção do título de Bacharel em Engenharia de Computação.
Área de Concentração: Inteligencia Computacional, Banco de Dados

USP – São Carlos
Novembro de 2023

Mussatto, Murilo
Customização de modelos de linguagem por meio de bancos de dados vetoriais / Murilo Mussatto. - São Carlos - SP, 2023.
53 p.; 29,7 cm.

Orientador: Jose Fernando Rodrigues Júnior.
Monografia (Graduação) - Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos - SP, 2023.

1. Customização de LLMs. 2. arquitetura RAG. 3. bases de dados vetoriais. 4. respostas a perguntas. I. Júnior, Jose Fernando Rodrigues. II. Instituto de Ciências Matemáticas e de Computação (ICMC/USP). III. Título.

Dedico esse trabalho aos meus queridos pais, fontes de inspiração e cujo amor incondicional foi essencial para minha trajetória pessoal e acadêmica.

AGRADECIMENTOS

A frase que melhor descreve o meu sentimento após realizar este trabalho foi escrita por Isaac Newton em 1676: "Se cheguei até aqui foi porque me apoiei nos ombros de gigantes".

Primeiramente, quero agradecer imensamente aos meus pais, Ercy e Lucimara, à minha irmã Livia e também aos meus avós, tios e tias que sempre olham por mim, mesmo que distantes. Tenho certeza que todo o suporte que recebi de minha família, desde o jardim de infância até o fim da faculdade, foi o que me fez chegar tão longe.

Além disso, agradeço a minha amada cachorrinha Lilica, companheira de incontáveis horas de estudo e dedicação. Tenho certeza que muitas vezes estava do meu lado por causa do ar-condicionado, no entanto sua presença e personalidade sempre me lembram do que é realmente importante na vida.

Agradeço também, de forma especial, à minha namorada Amanda, que durante inúmeras tardes e noites teve que me aguentar falando sobre os mais diversos assuntos enquanto estudava para provas e trabalhos. Não só isso, suas contribuições para esta monografia a transformaram em algo muito melhor do que qualquer coisa que eu poderia ter construído sozinho.

Faço também, um grande agradecimento aos meus amigos *eletrocompers*: Anakin, Bebedouro, Boy, Brunão, Chezão, Dois, Guerreiro, Ibitinga, João, Minion e Yudi. Amigos que fiz na faculdade, mas que levarei para toda a vida. Agradeço pelas brincadeiras, risadas, noites de jogatina e discussões calorosas sobre os assuntos mais banais existentes. Foram cinco anos de muitas batalhas onde nunca me senti sozinho e sempre pude contar com o apoio incondicional destes grandes guerreiros. Em particular, agradeço ao meu querido amigo Bruno, com quem tive a oportunidade de compartilhar diversos projetos da faculdade, incluindo a pesquisa desta monografia.

Por fim, agradeço ao professor José Fernando Rodrigues Júnior por ter aceitado orientar o trabalho desenvolvido nesta monografia. Agradeço também a todos os professores que fizeram parte de minha jornada na Universidade de São Paulo, muitas vezes me fazendo superar o que eu achava ser capaz fazer.

RESUMO

MUSSATTO, M.. **Customização de modelos de linguagem por meio de bancos de dados vetoriais**. 2023. 53 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Os grandes modelos de linguagem, ou *Large Language Models* (LLMs), têm revolucionado o campo da inteligência artificial ao transformar a forma como interagimos com a linguagem escrita e falada. Muitos modelos se mostraram fortemente capacitados para diversas tarefas de processamento de linguagem natural, como geração de texto, tradução, análise de sentimento, conversação com humanos, entre outros. Entretanto, a maioria destes modelos são pré-treinados e, por isso, seu conhecimento é limitado aos dados utilizados durante seu treinamento. Tendo em vista que treinar LLMs é relativamente custoso computacionalmente, pesquisadores buscaram criar formas de manipular a base de conhecimento destes modelos sem a necessidade de retreiná-los. Assim, desenvolveu-se uma arquitetura chamada *Retrieval Augmented Generation* (RAG), capaz de utilizar uma base de conhecimento externa para fornecer informações relvantes aos LLMs durante suas tarefas. Neste contexto, esta monografia tem como objetivo detalhar os passos necessários para implementar essa arquitetura, com foco na utilização de bases de dados vetoriais para armazenar essa base de conhecimento externa. Além disso, são citas algumas das principais ferramentas disponíveis no mercado que podem ser utilizadas para a implementação de cada um dos passos. Por fim, faz-se um pequeno experimento para demonstrar o funcionamento desta arquitetura com foco na tarefa de respostas a perguntas extrativas. Espera-se que este trabalho sirva como um guia para futuras implementações da arquitetura RAG e estimule a curiosidade do leitor para as vantagens que ela traz para o campo da inteligência artificial.

Palavras-chave: Customização de LLMs, arquitetura RAG, bases de dados vetoriais, respostas a perguntas.

ABSTRACT

MUSSATTO, M.. **Customização de modelos de linguagem por meio de bancos de dados vetoriais**. 2023. 53 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Large Language Models (LLMs) have revolutionized the field of artificial intelligence by transforming the way we interact with written and spoken language. Many models have proven to be highly capable in various natural language processing tasks, such as text generation, translation, sentiment analysis, conversation, among others. However, most of these models are pretrained, and therefore, their knowledge is limited to the data used during their training. Given that training LLMs is computationally costly, researchers have sought to create ways to manipulate the knowledge base of these models without the need to retrain them. Thus, an architecture called Retrieval Augmented Generation (RAG) was developed, capable of using an external knowledge base to provide relevant information to LLMs during their tasks. In this context, this monograph aims to detail the necessary steps to implement this architecture, with a focus on using vector databases to store this external knowledge base. Additionally, some of the main tools available in the market that can be used for the implementation of each step are mentioned. Finally, a small experiment is conducted to demonstrate the functioning of this architecture, with a focus on the task of extractive question answering. It is expected that this work serves as a guide for future implementations of the RAG architecture and stimulates the reader's curiosity about the advantages it brings to the field of artificial intelligence.

Key-words: LLM customization, RAG architecture, vector databases, question answering.

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

BERT .... *Bidirectional Encoder Representations for Transformers*
CNNs .... *Convolutional Neural Networks*
DeBERTa . *Decoding-enhanced BERT with Disentangled Attention*
GPT-3 .... *Generative Pre-trained Transformer 3*
IA ..... Inteligência Artificial
LLMs .... *Large Language Models*
MLM 2.0
NSP ..... *Next Sentence Prediction*
PLN ..... Processamento de Linguagem Natural
RAG ..... *Retrieval Augmented Generation*
RNN ..... *Recurrent Neural Networks*
RoBERTa . *Robustly Optimised BERT Approach*
SBERT ... Sentence-BERT
SQL ..... *Structured Query Language*
SQuAD .. *Stanford Question Answering Dataset*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Motivação e Contextualização

A Inteligência Artificial (IA) representa uma das áreas de pesquisa e desenvolvimento com maior ascensão nos tempos atuais, e seu impacto é significativo em uma variedade de setores, como saúde, educação, tecnologia da informação e entretenimento. Em particular, os grandes modelos de linguagem ou *Large Language Models* (LLMs), em inglês, desempenham um papel fundamental no panorama da IA e têm se destacado como catalisadores da revolução digital, transformando a forma como interagimos com a linguagem escrita e falada.

Uma das principais empresas do ramo de inteligência artificial é a OpenAi, que, em junho de 2020, lançou o *Generative Pre-trained Transformer 3* (GPT-3) (BROWN *et al.*, 2020). Este modelo de linguagem com aprendizado profundo representou um marco na área de aprendizado de máquina e processamento de linguagem natural por ser um dos maiores modelos já criados, com 175 bilhões de parâmetros. Essa enorme quantidade de parâmetros torna o modelo capaz de compreender contextos complexos e de gerar textos de alta qualidade para a execução de diferentes tarefas, incluindo tradução, geração, respostas a perguntas, entre outros.

Pouco tempo depois, a OpenAi apresentou para o mundo outro modelo que revolucionou a percepção do público geral sobre inteligência artificial. Trata-se do ChatGPT (OPENAI, 2023), uma variação do GPT-3 projetada especificamente para interações em linguagem natural, como conversas interativas frequentemente presentes em *chatbots* e assistentes virtuais. O lançamento destes dois modelos trouxe a atenção do público e da comunidade de pesquisa para as capacidades cada vez maiores dos modelos de linguagem com aprendizado profundo. Além da OpenAi, diversas outras empresas, como Google e Meta, têm contribuído para o desenvolvimento da inteligência artificial, por meio do lançamento de modelos próprios de linguagem como o BARD (PICHAI, 2023) e o LLaMA (TOUVRON *et al.*, 2023), referente às respectivas empresas citadas.

Todos os modelos citados até então possuem algo em comum: são todos modelos pré-treinados. Ou seja, são modelos que passaram por um extenso e custoso processo de treinamento, utilizando uma grande quantidade de dados que possibilita a realização de suas funções. Entretanto, os modelos em questão possuem duas grandes desvantagens, ambas relacionadas ao conjunto de dados utilizados para treiná-los. A primeira delas é a rigidez com que sua base de conhecimento é introduzida no modelo. Os LLMs têm um conhecimento limitado com base no que foi usado em seu treino até a sua data de corte. Isso significa que, para adicionar algum dado

ao modelo, é preciso treiná-lo novamente. Esta é uma operação que requer recursos computacionais substanciais, o que pode ser de alto custo e inacessível para muitos desenvolvedores e organizações. Além disso, por estarem presos ao conjunto de dados utilizado durante seu treinamento, os LLMs podem refletir e amplificar os preconceitos presentes nestes dados, o que pode resultar em respostas ou geração de textos sexistas, racistas ou tendenciosos em diversas formas. A segunda desvantagem desses modelos é que eles são suscetíveis a alucinações quando não possuem conhecimento suficiente para gerar uma resposta. Isso, pois os LLMs geram texto com base nas probabilidades estatísticas aprendidas durante a fase de treinamento, e, quando confrontados com informações ambíguas ou ausentes em sua base de conhecimento, podem gerar informações falsas ou sem sentido.

No ano de 2020, como uma possível solução para as desvantagens dos modelos de linguagem pré-treinados, a empresa Meta, na época conhecida como Facebook, apresentou um modelo de arquitetura chamado *Retrieval Augmented Generation* (RAG) (LEWIS *et al.*, 2020). A principal proposta dessa arquitetura é combinar memória paramétrica pré-treinada com memória não paramétrica recuperada sob demanda. Isto é, além da base de conhecimento introduzida no LLM durante a fase de treinamento (memória paramétrica pré-treinada), na arquitetura RAG é adicionado ao modelo uma nova base de conhecimento (memória não paramétrica) que pode ser utilizada para melhorar as respostas geradas.

Assim, a arquitetura RAG atua ampliando a precisão de modelos que funcionam segundo o padrão "seq2seq"(SUTSKEVER; VINYALS; LE, 2014), o qual recebe uma sequência de entrada e a transforma em outra sequência em sua saída. Porém, existe um passo adicional que diferencia e eleva a arquitetura RAG perante os outros modelos. Ao invés de passar a sequência de entrada diretamente para o gerador de textos, essa arquitetura usa a entrada para recuperar um conjunto de documentos relevantes, geralmente com fatos, definições e conceitos, que são concatenados com a entrada original e servem de contexto para o modelo gerar a sequência de saída. Dessa forma, é possível gerar textos mais relevantes e corretos, inclusive diminuindo drasticamente as alucinações produzidas pelo modelo, já que as respostas serão baseadas em um conjunto de dados externos.

Essa nova base de conhecimento, a não-paramétrica, pode ser introduzida de diversas formas, como através de um vetor indexado contendo artigos do site Wikipédia, exemplo explorado por Lewis *et al.* (2020). Outra possível forma de adicionar esses conhecimentos ao modelo é por meio de bases de dados vetoriais como o ChromaDB (CHROMADB, 2023) ou o PGVector (PGVECTOR, 2023), uma extensão para o famoso gerenciador de banco de dados PostgreSQL (POSTRESQL, 2023). Isso promove uma grande vantagem para a arquitetura RAG, já que os modelos não precisam ser "retreinados". Caso seja necessário alterar seus conhecimentos, basta remover ou adicionar documentos presentes na base externa, sendo esta uma operação muito mais simples em termos de complexidade e uso de recursos computacionais se comparado a um novo treinamento.

Dessa forma, ficam claras as vantagens da arquitetura RAG, tanto na diminuição de alucinações produzidas, como na flexibilidade para alterar a base de conhecimento do modelo sendo utilizado. Após a apresentação dessa arquitetura, muitos pesquisadores e desenvolvedores se propuseram a criar diferentes formas de implementar essa arquitetura, buscando flexibilidade e adaptabilidade para resolver os mais diversos problemas. Neste contexto, esta monografia busca compilar os passos necessários para implementar essa arquitetura, utilizando uma base de dados vetorial para armazenar informações adicionais. Além de promover uma detalhada explicação teórica, busca-se também demonstrar algumas das principais ferramentas que podem ser utilizadas na implementação de um modelo simples do tipo RAG, com foco na tarefa de perguntas e respostas.

1.2 Objetivos

O objetivo é prover uma introdução aos conceitos necessários para que seja possível customizar um LLM. Assim, este poderá exercer a função de responder perguntas sobre um conjunto de dados com base na arquitetura RAG. Para alcançar esse objetivo, primeiro será feita uma exposição teórica sobre conceitos, como *embeddings*, bases de dados vetoriais, e LLMs, necessários para entender o surgimento e a construção dessa arquitetura. Em seguida, serão detalhados os passos necessários para a implementação de uma arquitetura RAG em um modelo simples. Estes passos são:

1. Quebra de documentos em pedaços (*chunks*);
2. Transformação dos *chunks* em *embeddings*;
3. Armazenamento dos vetores criados em uma base de dados vetoriais;
4. Busca por similaridade dos vetores mais relevantes perante uma dada entrada;
5. Inserção de documentos como contexto para um LLM;

Em cada passo, serão destacados diferentes métodos para a realização de cada tarefa, juntamente de suas vantagens, desvantagens, e *trade-offs*. Além disso, serão apresentadas diferentes ferramentas que podem ser utilizadas para a execução de cada passo. Por fim, serão escolhidas algumas dessas ferramentas para a implementação de um modelo simples, capaz de responder perguntas sobre um texto retirado de um artigo referente ao Brasil, no site Wikipédia (WIKIPÉDIA, 2023). Para esse experimento, serão utilizados dois LLMs para responder a cinco perguntas em relação ao texto, e as respostas obtidas serão comparadas e analisadas para avaliar o desempenho da implementação.

Dessa forma, espera-se que este trabalho sirva como guia para uma possível implementação da arquitetura RAG em grandes modelos de linguagem. Assim, por meio da utilização de

bases de dados vetoriais para o armazenamento de documentos externos, é possível demonstrar a flexibilidade e a melhoria proporcionada por essa arquitetura. Com isso, espera-se que a comparação realizada traga à tona possíveis espaços para melhorias no campo dos grandes modelos de linguagem.

1.3 Organização

No Capítulo 2, é apresentada uma revisão da terminologia dos conceitos sobre os quais é construída a arquitetura RAG, bem como os trabalhos da literatura relacionados a ela. Em seguida, no Capítulo 3, são descritos os passos para implementar esta arquitetura, detalhando vantagens e desvantagens das abordagens para cada passo, além de ferramentas que podem ser utilizadas para a implementação de um modelo simples baseado em RAG. Finalmente, no Capítulo 4, apresentam-se as conclusões sobre a arquitetura RAG e a implementação desenvolvida para a tarefa de respostas a perguntas. Além disso, comenta-se sobre trabalhos futuros que podem aprimorar ainda mais o desempenho de modelos de linguagem em tarefas intensivas em conhecimento.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

2.1 Considerações Iniciais

Neste capítulo, serão abordados os conceitos e terminologias necessários para entender o funcionamento da arquitetura RAG. Além disso, serão detalhadas algumas ferramentas disponíveis para a implementação dessa arquitetura, como o *framework* "Langchain"(CHASE, 2022) e a biblioteca em python "Trasformers"(WOLF *et al.*, 2020).

2.2 Grandes Modelos de Linguagem (LLMs)

Os modelos de linguagem trabalham analisando a probabilidade de sequências de palavras serem válidas ou inválidas. Essa tarefa é muito recorrente em diversos campos do Processamento de Linguagem Natural (PLN), como tradução e geração de textos, análise de sentimento, correção ortográfica, entre outros. Para realizar essa análise, muitos métodos foram desenvolvidos ao longo do tempo, como é o caso das redes neurais recorrentes, ou *Recurrent Neural Networks* (RNN) e das redes neurais convolucionais ou *Convolutional Neural Networks* (CNNs), que pavimentaram o caminho para o treinamento de modelos de linguagem mais complexos. No entanto, foi com o advento das arquiteturas de atenção, como o *transformer*, que os modelos de linguagem alcançaram um avanço notável em desempenho e capacidade.

Introduzida no artigo "*Attention Is All You Need*"(VASWANI *et al.*, 2017), a arquitetura *transformer* trouxe várias vantagens em relação às abordagens anteriores. Em especial, pode-se destacar a capacidade de paralelismo desta arquitetura, permitindo o processamento simultâneo de palavras. Isto torna este modelo muito mais rápido na realização de tarefas, se comparado aos métodos anteriores. Ademais, através do processo de ajuste fino, é possível utilizar essa arquitetura, ou até mesmo partes dela, para as mais diversas tarefas.

Os *transformers* podem ser divididos em duas partes, os *encoders* (codificadores) e os *decoders* (decodificadores). A primeira é responsável por analisar palavras de um texto e transformá-las em representações vetoriais multidimensionais, os *embeddings*. Já a segunda parte, é responsável pelo processo inverso, ou seja, transformar os *embeddings* novamente em uma forma textual legível para humanos. Assim, estas duas partes estão fortemente relacionadas, mas é possível utilizar apenas uma delas, dependendo da tarefa em questão. Por exemplo, para a

classificação de textos, os *encoders* são mais utilizados pois as representações vetoriais criadas por eles podem ser utilizadas por classificadores de texto.

No contexto da arquitetura RAG, tanto os *encoders* quanto os *decoders* são muito utilizados, como será analisado no decorrer desta monografia. Isso pois, a base de conhecimento externa geralmente é armazenada no formato de *embeddings* que permitem a recuperação de documentos relevantes com base na semelhança a uma entrada do usuário. Além disso, para a introdução destes documentos em um LLM, é necessário transformá-los novamente em sua forma textual.

2.2.1 BERT, RoBERTa e DeBERTa

Dentre os LLMs que utilizam a arquitetura *transfomers*, o *Bidirectional Encoder Representations for Transformers* (BERT) (DEVLIN *et al.*, 2018) é um dos mais famosos. Ele foi desenvolvido por pesquisadores da Google em 2018 e é um dos modelos mais versáteis que existem, trazendo soluções para as tarefas mais comuns de processamento de linguagem. Uma das principais inovações trazidas pelo BERT está na forma como este modelo foi pré-treinado, realizando duas tarefas relativamente simples simultaneamente. A primeira delas é o (MLM)Masked Language Model onde o modelo é alimentado com diversos textos que possuem algumas de suas palavras escondidas, as quais ele deve descobrir como eram no texto original. A segunda tarefa é o*Next Sentence Prediction* (NSP) onde duas frases eram fornecidas ao BERT e este deve prever se a segunda frase segue a primeira no texto original ou se elas não são relacionadas.

Além disso, outro diferencial do modelo BERT é o seu pré-treinamento em um contexto bidirecional, no qual ele consegue entender a relação tanto entre as palavras que o precedem quanto as que o sucedem em uma sentença. Isso permite que o modelo capture nuances semânticas e contextuais de maneira mais precisa, o que não era possível em outros modelos que levavam em consideração apenas palavras anteriores à que está sendo analisada.

Pouco tempo depois do desenvolvimento do modelo BERT, em 2019, pesquisadores do Facebook desenvolveram o *Robustly Optimised BERT Approach* (RoBERTa) (LIU *et al.*, 2019). Este é um modelo aprimorado que usa a mesma arquitetura do BERT mas traz melhorias significativas na etapa de pré-treinamento. Uma destas melhorias foi utilizar um "mascaramento dinâmico", em que, para um determinado texto, diferentes palavras eram escondidas a cada vez que o modelo era treinado. Assim, ao modificar as palavras que eram mascaradas, o modelo teve que entender melhor o contexto do trecho para poder descobrir a palavra original. Além disso, o modelo RoBERTa também foi treinado com sequências de texto maiores do que as utilizadas para o BERT, o que permitiu ao modelo aumentar seu desempenho em tarefas de PLN.

Um ano após a apresentação do modelo do Facebook, pesquisadores da Microsoft propuseram outro modelo, o *Decoding-enhanced BERT with Disentangled Attention* (DeBERTa)

(HE *et al.*, 2020). Este modelo trouxe melhoras significativas de desempenho, se comparado aos modelos anteriores, devido a utilização de duas novas técnicas: o *disentangled attention mechanism* (mecanismo de atenção desembaraçado, em português) e o *enhanced mask decoder* (decodificador de máscaras aprimorado, em português). A primeira técnica envolve a utilização de dois vetores para representar cada palavra, um para o conteúdo e outro para a posição. Ao contrário do modelo BERT, que utiliza apenas um vetor para representar cada palavra, o DeBERTa faz o uso de matrizes para computar o conteúdo com relação à posição de cada palavra no texto. Essa técnica permite ao modelo entender que palavras como "aprendizado"e "máquina"possuem uma dependência muito mais forte quando são localizadas lado a lado, como em "aprendizado de máquina". Já a segunda técnica, permite que o DeBERTA utilize a posição absoluta de uma palavra em uma frase na tarefa de MLM. Isso porque, em muitos casos, o significado sintático de uma palavra está atrelado à sua posição absoluta dentro da frase, como é o caso da identificação de sujeitos em uma oração.

Estes três modelos de linguagem foram destacados, porque terão um papel significativo nesta monografia. Os modelos BERT e RoBERTa foram utilizados como base para criar o SBERT (Sentence BERT), um modelo capaz de gerar *embeddings* de frases completas, que será explorado na seção 3.3 - Criação de *Embeddings*. Já em relação ao modelo DeBERTa, uma variação multi-linguística foi utilizada para o desenvolvimento da implementação de um modelo RAG, que será detalhado na seção 3.5 - Inserção dos documentos como contexto para LLMs.

2.3 Langchain e biblioteca Transformers

O framework Langchain (CHASE, 2022), lançado em outubro de 2022, foi desenvolvido para facilitar a construção de aplicações baseadas em grandes modelos de linguagem. Ele oferece uma série de interfaces e métodos auxiliares além de integrações com outros serviços que permitem unir o poder dos LLMs com grandes quantidades de dados e outras aplicações. Isto o torna uma ferramenta poderosa que pode ser utilizada para agilizar o processo de construção de uma aplicação ou torná-la mais robusta.

No contexto da arquitetura RAG, o Langchain oferece muitas facilidades. Por exemplo, uma categoria de métodos auxiliares muito útil são os *document loaders*, que são capazes de extrair textos de diversas fontes como arquivos, websites e códigos fonte. Assim, torna-se simples construir a base de conhecimento externa; basta extrair os textos das fontes desejadas, utilizando as funções prontas oferecidas. Outro benefício são as integrações com diversas bases de dados, o que facilita o armazenamento e recuperação dos documentos que foram extraídos.

Já a biblioteca Transformers (WOLF *et al.*, 2020), para a linguagem de programação Python, foi desenvolvida para facilitar o treinamento e utilização de grandes modelos de linguagem. Esta biblioteca permite a conexão com o repositório de modelos do site HugginFace(HUGGINFACE, 2023a), uma coleção de milhares de LLMs, baseados na arquitetura

transformers. Neste site, é possível baixar desde modelos completamente treinados até modelos que precisam passar pelo processo de ajuste fino. Além disso, a biblioteca Transformers unifica todos os modelos disponíveis sob uma única interface, tornando extremamente fácil substituir um LLM por outro.

Essa unificação se dá por meio de métodos chamados *pipelines*, que simplificam a utilização dos LLMs com base na tarefa que se deseja realizar. Cada tarefa possui sua própria *pipeline*, como a *SummarizationPipeline* para sumarização de textos, a *TextGenerationPipeline* para geração de texto e a *QuestionAnsweringPipeline* para a tarefa de respostas a perguntas. Para utilizar uma *pipeline* desta biblioteca, basta indicar qual a tarefa está sendo realizada e passar o identificador do LLM que será utilizado, o qual pode ser retirado do próprio site do HugginFace.

Finalmente essas duas tecnologias se mostram muito adequadas para a implementação da arquitetura RAG. O Langchain permite a extração de documentos para compor a base de conhecimento, assim como a utilização de bases de dados para armazená-los. Já a biblioteca Transfomers, permite a utilização, de forma simples, de inúmeros modelos de linguagem para a realização das mais diversas tarefas. Assim, estas ferramentas são comentadas durante o capítulo 3 - Desenvolvimento, conforme são utilizadas na aplicação desenvolvida.

2.4 Trabalhos Relacionados

Como trabalhos relacionados, é possível citar o próprio artigo que primeiro apresentou a arquitetura "RAG: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"(LEWIS *et al.*, 2020). Este artigo, escrito por pesquisadores do Facebook, descreve em detalhes o funcionamento da arquitetura além de citar alguns experimentos desenvolvidos para colocar em prática o que foi apresentado.

Ademais, é possível citar o artigo "Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast"por Topsakal e Akinci (2023). Este artigo explora as principais funções presentes no núcleo do framework Langchain para o desenvolvimento de aplicações utilizando grandes modelos de linguagem.

2.5 Considerações Finais

Após este capítulo, espera-se que o leitor tenha adquirido familiaridade com os grandes modelos de linguagem baseados na arquitetura *transformer*, em especial os modelos BERT, RoBERTa e DeBERTa. Além disso, espera-se ter estimulado a curiosidade sobre as ferramentas Langchain e a biblioteca Transformers, muito utilizadas na implementação de modelos RAG.

No próximo capítulo é feita uma exploração teórica sobre os passos necessários para implementar a arquitetura RAG utilizando bases de dados vetoriais. São detalhadas diversas abordagens e pontos a serem levados em conta durante a implementação de cada um dos

passos. Além disso, são citadas algumas das ferramentas disponíveis no mercado que podem ser utilizadas para a construção de uma aplicação.

Por fim, são apresentados os códigos-fonte que fizeram parte da implementação da arquitetura RAG, desenvolvida neste trabalho. Cada uma das funções utilizadas é explicada, assim como o motivo de sua escolha.

Capítulo 3

DESENVOLVIMENTO

3.1 Considerações Iniciais

Neste capítulo, são apresentados os passos necessários para implementar a arquitetura RAG sobre um LLM com o objetivo de realizar tarefas de respostas a perguntas. A implementação descrita pode ser dividida em quatro passos:

1. Quebra dos documentos em *chunks*;
2. Transformação dos *chunks* em *embeddings*;
3. Armazenamento dos vetores criados em uma base de dados vetorial;
4. Busca por similaridade dos vetores mais relevantes perante uma entrada;
5. Inserção dos documentos recuperados como contexto para um LLM.

Cada passo é apresentado em mais detalhes nas seções a seguir. São exploradas as abordagens existentes para realizar cada tarefa, juntamente das ferramentas e tecnologias disponíveis. Por fim, é realizada uma implementação da arquitetura usando-se algumas das abordagens descritas. O objetivo é demonstrar o funcionamento da arquitetura RAG na tarefa de respostas a perguntas. O modelo desenvolvido utiliza, como base de conhecimento externa, um texto retirado da página ”Brasil” do site Wikipédia (WIKIPÉDIA, 2023). Assim, são feitas cinco perguntas referentes ao texto utilizado e as respostas obtidas são analisadas para avaliar o desempenho dos modelos.

3.2 Quebra dos documentos em *chunks*

O primeiro passo para implementar um modelo baseado em RAG é tratar a base de conhecimento. Esse tratamento se dá pela quebra dos documentos em diversos fragmentos, chamados *chunks*, que serão futuramente transformados em vetores numéricos, os *embeddings*. A quantidade de informação presente em cada fragmento depende da forma como texto é divido, podendo variar de apenas algumas frases até vários parágrafos do texto original. Este processo de divisão é chamado de *chunking*, uma etapa crucial para obter bons resultados, independente da tarefa alvo da implementação.

Para que um documento seja processado por um modelo de linguagem, ele deve ser, primeiramente, dividido em *tokens*, que representam uma unidade básica de texto. O que define exatamente um *token* é o algoritmo utilizado para fazer essa divisão, mas geralmente eles são caracteres, palavras, subpalavras ou qualquer outro segmento de texto. Neste contexto, muitos LLMs possuem uma limitação para o número de *tokens* que podem ser utilizados como contexto para a geração de texto. Ou seja, textos muitos grandes podem não ser processados por inteiro ou, no pior caso, rejeitados completamente devido ao excesso de *tokens* presentes. Por esse motivo, é possível definir um tamanho limite para cada *chunk*, prevenindo os casos em que os documentos processados excedem o limite de *tokens* imposto pelo modelo de linguagem.

A quebra dos documentos auxilia no processo de criação dos *embeddings*, gerando vetores mais otimizados para cada tarefa específica. Quando os documentos são quebrados, é possível escolher métodos que identificam e extraem unidades semânticas relevantes, como frases, nomes e informações gramaticais. Essas unidades semânticas serão transformadas em vetores que mais bem representam algum conceito, fato ou informação presente no texto. Deste modo, quando for preciso recuperar documentos com base em uma entrada fornecida pelo usuário, será mais fácil encontrar *chunks* que se relacionam com o que foi fornecido. Com isso, o processo de *chunking* promove uma maior eficiência de processamento, uma vez que o LLM lida com unidades menores de texto, o que reduz a carga computacional e melhora o desempenho do modelo ao processar o contexto introduzido.

Neste cenário, uma das decisões a ser tomada é definir o tamanho ideal para cada *chunk*, isso pois existem várias vantagens e desvantagens em usar fragmentos maiores ou menores. Por um lado, *chunks* maiores apresentam uma maior quantidade de informações relevantes que podem ser utilizadas pelo LLM para gerar uma resposta mais completa. Porém, quanto maior for o tamanho do fragmento, mais tempo e poder de processamento será necessário para analisá-lo. Por outro lado, *chunks* menores oferecem mais agilidade nas respostas, mas apresentam um grande risco: informações essenciais e parte do contexto de onde estão sendo retirados podem não estar presentes nos principais *chunks* recuperados. Assim, é necessário avaliar cuidadosamente o tipo de base de conhecimento utilizada na implementação do RAG para, então, escolher o tamanho de *chunk* mais adequado para a tarefa que será realizada.

Existem várias métricas que podem ser empregadas para selecionar o tamanho de *chunk*. Uma delas é analisar o tipo de texto que compõe a base de conhecimento. No caso de textos grandes, como artigos e livros, é ideal escolher tamanhos menores para fazer o *chunking*, pois este tipo de texto, geralmente, é composto por diversas ideias e conceitos que podem ser mais bem compreendidos de forma separada. Caso os textos sejam pequenos, como mensagens e emails, vale a pena utilizar tamanhos maiores, já que uma menor variedade de informações estão presentes nesses tipos de texto.

Outra forma que pode auxiliar na definição do tamanho dos *chunks* é analisar o tipo de *embedding* que será criado a partir deles. Existem diversos métodos para transformar um

pedaço de texto em um vetor numérico, como será explorado na próxima seção; cada um deles se beneficia de um tipo de divisão específica. Por exemplo, um transformador de sentenças pode produzir *embeddings* de alta qualidade quando utilizado em frases, mas pode perder qualidade caso o texto contenha mais conteúdo.

A complexidade das entradas fornecidas pelo usuário é outro fator que pode influenciar no tamanho dos *chunks*. Se o usuário fornecer entradas curtas ou estiver à procura de informações específicas, como o significado de uma palavra no dicionário, é mais interessante utilizar *chunks* menores, que serão mais rápidos e precisos. Caso o usuário faça requisições mais complexas envolvendo diversos conteúdos, como pedir o resumo de um livro, é melhor utilizar *chunks* maiores, que englobam diversas frases e até parágrafos inteiros, pois a quantidade de informação requerida para gerar uma boa resposta também é maior.

Em relação aos métodos utilizados para a realização do *chunking*, existem três principais (SCHWABER-COHEN, 2023). O primeiro e mais simples é chamado de ”*chunking* de tamanho fixo” (do inglês, *“Fixed-Size Chunking”*), que consiste na escolha de um número fixo de *tokens* para compor o tamanho de cada *chunk*. Assim, o texto é dividido sequencialmente, do começo ao fim, enquanto cria novos *chunks* quando o número de *tokens* exceder o limite escolhido. Ocasionalmente é possível utilizar a sobreposição de *chunks*, em que parte dos *tokens* está presente em dois fragmentos adjacentes, o que garante que o texto todo será dividido em partes de tamanho igual. Este método possui ótima eficiência computacional, pois não depende de bibliotecas complexas de processamento de linguagem para funcionar.

O segundo método é chamado de”*chunking* sensível ao conteúd” (do inglês *“Content-Aware Chunking”*) e leva em consideração o conteúdo que está sendo dividido, fazendo com que os *chunks* possam ter tamanhos diferentes entre si. Este processo pode envolver a identificação de estruturas lógicas dentro dos dados, como parágrafos em um texto, tabelas em um documento ou transições entre diferentes tópicos. Esse método ajuda a preservar a integridade do conteúdo e facilita o processamento subsequente. Fora isso, este tipo de divisão pode facilitar na recuperação e pesquisa de informações, uma vez que os fragmentos são mais relevantes para as consultas, quando comparados a pedaços de tamanho fixo. O ”*chunking* sensível ao conteúdo” é o método mais indicado para a implementação de uma arquitetura RAG, pois preserva as informações da base de conhecimento externa. Entretanto, o custo computacional para fazer essa divisão é elevado, já que é necessário analisar o conteúdo enquanto este está sendo dividido. Além disso, existem formas de dividir o texto mantendo as unidades semânticas agrupadas, mas deixando de realizar o processamento do conteúdo, como será explorado a seguir.

O terceiro método, chamado de "*chunking* recursivo", é similar ao primeiro no sentido de que busca-se criar *chunks* de tamanhos definidos. Porém, enquanto no *chunking* de tamanho fixo o texto é lido sequencialmente, sempre criando novos fragmentos quando o número de *tokens* excede um limite estabelecido, nesta abordagem o texto é divido recursivamente até que os *chunks* criados sejam menores que o tamanho limite. Além disso, utilizando este método

é possível dividir o texto com base em caracteres de referência, como pontos finais e quebras de linha, o que permite que unidades semânticas, como frases e parágrafos, sejam mantidas nos fragmentos criados. Assim, esse método é o mais utilizado em implementações do modelo RAG já que permite separar o texto de forma semelhante ao segundo método e com um custo computacional mais baixo.

Para a implementação desenvolvida neste trabalho, tentou-se primeiro utilizar um método do *framework* LangChain, chamado *"WebBaseLoader"*, para baixar o texto contido na página do Wikipédia, diretamente da web. Entretanto, notou-se que essa ferramenta não fez uma boa extração, principalmente quanto a codificação de alguns caracteres da língua portuguesa. Assim, optou-se por copiar manualmente o texto em questão e disponibilizá-lo através de um arquivo, armazenado em um repositório do GitHub (MUSSATTO, 2023). Deste modo, o primeiro passo no desenvolvimento foi utilizar o método *"TextLoader"*, também disponibilizado pelo Langchain, para extrair o texto do arquivo "brasil.txt".

O próximo passo foi realizar a quebra do texto em vários fragmentos. Para isso, o *framework* possui funções para realizar cada um dos métodos de *chunking* comentados durante a seção. Como comentado anteriormente, o *chunking* recursivo é um dos algoritmos mais utilizados para implementações da arquitetura RAG . Assim, utilizou-se a função *"RecursiveCharacterTextSplitter"* para aplicar este método de fragmentação no texto em questão.

Como pode ser observado no código 1, o tamanho definido para os *chunks* criados foi de 500 *tokens*, gerando um total de **242 fragmentos**. Por ser um texto relativamente pequeno (apenas uma página da web) mas repleto de diferentes conceitos, era preciso encontrar um tamanho de *chunk* grande o suficiente para englobar pelo menos uma frase completa, mas não tão grande tal que o número de fragmentos gerados fosse demasiadamente pequeno. Por processo de tentativa e erro, foi concluído que o número de *tokens* escolhido era um bom tamanho para a implementação.

Além disso, analisando o código é possível notar que os caracteres escolhidos para fazer a separação foram: *"\n\n"* (indicador de quebra de parágrafo), *"\n"* (indicador de quebra de linha) e *"."* (ponto final). Isso permitiu a criação de *chunks* que mantiveram intactas as unidades semânticas do texto original. Alguns exemplos dos fragmentos criados são:

- "É o único país na América onde se fala majoritariamente a língua portuguesa e o maior país lusófono do planeta, além de ser uma das nações mais multiculturais e etnicamente diversas, em decorrência da forte imigração oriunda de variados locais do mundo. Sua atual Constituição, promulgada em 1988, concebe o Brasil como uma república federativa presidencialista, formada pela união dos 26 estados, do Distrito Federal e dos 5 571 municípios."

- "Na época colonial, cronistas da importância de João de Barros, frei Vicente do Salvador e Pero de Magalhães Gândavo apresentaram explicações concordantes acerca da origem do

nome "Brasil". De acordo com eles, o nome "Brasil"é derivado de "pau-brasil", designação dada a um tipo de madeira empregada na tinturaria de tecidos."

Código-fonte 1: Extração e quebra do texto em *chunks*

```
1 from langchain.document_loaders import TextLoader
2 from langchain.text_splitter import RecursiveCharacterTextSplitter
3
4 # Document Loader
5 loader1 = TextLoader('./Wikipedia-Brasil-Page/brasil.txt')
6 documents = loader1.load()
7
8 # Text Splitter
9 text_splitter = RecursiveCharacterTextSplitter(
10     chunk_size=500,
11     length_function = len,
12     is_separator_regex = False,
13     separators=["\n\n", "\n", "."]
14 )
15 chunks = text_splitter.split_documents(documents)
16 print(len(chunks))
```

Após se concluir o processo de *chunking*, é preciso transformar cada um dos pedaços criados em representações vetoriais numéricas, os *embeddings*. Este processo é necessário para que seja possível aplicar algoritmos e fórmulas matemáticas para recuperar documentos com base nas semelhanças semânticas entre eles. Na próxima seção, são exploradas algumas das diversas abordagens para a criação dessas representações vetoriais.

3.3 Criação de *Embeddings*

Embeddings são representações vetoriais de dados comumente utilizados para representar palavras, frases, documentos ou até mesmo entidades em um espaço multidimensional. Essas representações vetoriais são especialmente úteis, porque capturam informações semânticas e relações entre os dados, tornando-as adequadas para a realização de tarefas de análise de texto e modelagem de linguagem. Além disso, a representação de uma informação em um vetor numérico permite que uma série de abordagens matemáticas sejam aplicadas sobre eles, como, por exemplo, calcular a distância entre dois *embeddings* pode indicar se eles representam conceitos próximos semanticamente. No contexto da arquitetura RAG, os *embeddings* são cruciais para sua implementação por permitirem que apenas informações relevantes à entrada fornecida pelo usuário sejam utilizadas para compor o contexto do LLM. Dessa forma, a escolha de certa metodologia para a criação destas representações vetoriais impacta diretamente no desempenho da arquitetura implementada. Se um algoritmo de baixa qualidade for utilizado,

informações não relevantes ou incompletas podem ser utilizadas para gerar respostas parciais, inconclusivas ou até mesmo incorretas. Assim, é de extrema importância escolher um algoritmo de maior qualidade.

O *embedding* de sentenças captura o sentido de uma frase inteira considerando o contexto de cada palavra. Os *sentence embeddings* permitem que informações sejam agrupadas com base no seu significado e no conceito sendo descrito. Esse agrupamento é muito útil para implementações da arquitetura RAG, pois a base de conhecimento pode ser quebrada em diversos vetores, em que cada um representa alguma informação ou conceito específico. Esses vetores, então, podem ser comparados semanticamente com alguma entrada do usuário, e um algoritmo pode decidir quais partes da base de conhecimento são interessantes e quais partes não tem relação ao que foi fornecido pelo usuário.

Assim como nos *embeddings* de palavras, existem diversos algoritmos capazes de criar representações vetoriais de frases completas. Um deles é o "Skip-Thought Vectors", proposto em 2015 por Kiros *et al.* (2015), cuja principal característica é ser uma extensão do modelo "Word2Vec"(MIKOLOV *et al.*, 2013), que treina modelos para prever sentenças vizinhas em um grande corpus de texto. Além dele, outra abordagem muito comum é utilizar modelos de linguagem baseados em *transformers*, como o BERT - que gera *embeddings* de palavras -, e fazer a média de todos os vetores gerados para uma determinada frase. Ambas as abordagens possuem grandes desvantagens. Enquanto a primeira é muito rígida e não permite o ajuste fino para tarefas de PLN, algo permitido em modelos de linguagem como o BERT, a média dos *word embeddings* não captura a ordem das palavras e a estrutura sintática da frase, o que torna os vetores criados pouco precisos. Dessa forma, novas abordagens precisaram ser criadas para o desenvolvimento de *embeddings* de sentenças. Devido a isso, em 2019 desenvolveu-se o Sentence-BERT (SBERT) (REIMERS; GUREVYCH, 2019), que permite utilizar a flexibilidade do modelo de linguagem BERT para gerar *sentence embeddings* de altíssima qualidade.

O SBERT foi desenvolvido para solucionar o problema de comparação entre duas sentenças. Muitos modelos de linguagem podem ser treinados para realizar esse tipo de tarefa, até mesmo o BERT. Entretanto, mesmo em modelos tão avançados, esta não é uma tarefa simples computacionalmente. Por exemplo, o modelo BERT trabalha com um codificador cruzado, ou seja, duas frases são inseridas na rede *transformer* e um valor de similaridade entre elas é gerado. Dessa forma, achar o par de frases mais semelhantes em uma coleção de $n = 10000$ frases utilizando o BERT, requer $n \cdot (n-1)/2 = 49995000$ computações, o que representa 65 horas em uma GPU moderna, como explica Reimers e Gurevych (2019).

Assim, transformar as frases em vetores e utilizar fórmulas matemáticas para encontrar a similaridade entre elas é uma abordagem muito mais promissora. Contudo, tentar criar *sentence embeddings* utilizando a média dos *word embeddings* gerados pelo modelo BERT, se provou pouco eficaz, como foi discutido anteriormente. Dessa maneira, o SBERT foi criado como uma modificação do modelo BERT, utilizando estruturas de redes siamesas e de redes de triplos

(*siamese and triplet network structures*, em inglês) em conjunto para gerar *sentence embeddings* que podem ser comparados utilizando o algoritmo de similaridade por cosseno.

A técnica conhecida como rede siamesa, empregada pelo SBERT, envolve o uso de duas redes neurais idênticas, no caso, dois modelos BERT que compartilham os mesmos parâmetros. Essas redes gêmeas processam pares de frases de entrada com o objetivo de gerar representações semânticas robustas para cada uma delas. Além disso, durante a etapa de treinamento do SBERT, são utilizados triplos de sentenças, que consistem em três partes: a âncora, a positiva e a negativa. A sentença âncora é aquela usada como a referência para a qual se deseja criar uma representação semântica. Já a sentença positiva é escolhida de modo a ser semanticamente similar à âncora, compartilhando o mesmo contexto ou significado. A sentença negativa, por outro lado, é selecionada para ser semanticamente diferente da âncora. Com isso, o processo de treinamento do SBERT é projetado para fazer com que as representações da sentença âncora e da sentença positiva fiquem próximas uma da outra, enquanto, ao mesmo tempo, afasta a representação da sentença âncora da sentença negativa. A combinação dessas duas técnicas durante seu treinamento faz com que o SBERT seja capaz de gerar *embeddings* semânticos para qualquer frase de entrada. Os vetores criados podem ser, então, comparados utilizando uma função de similaridade com cosseno para determinar quão próximos são entre si. Essa capacidade de gerar *sentence embeddings* torna o SBERT um modelo valioso em várias aplicações, em especial na determinação da similaridade semântica entre sentenças e na recuperação de informações relevantes, tarefas que estão no núcleo de uma arquitetura RAG.

Deste modo, foi utilizado um modelo de *sentence embedding*, baseado no SBERT, para a transformação dos *chunks* criados na etapa anterior. Em seu website, é possível ver um comparativo entre os modelos pré-treinados disponibilizados (SBERT.NET, 2023). Como o texto da base de conhecimento está em português, foi preciso escolher um modelo que conseguisse entender a linguagem sendo utilizada. Assim, o modelo escolhido foi o ***"distiluse-base-multilingual-cased-v1"*** (HUGGINFACE, 2023c). Ele foi escolhido por permitir a criação de *embeddings* a partir de textos de diversas línguas, como o português, o inglês, o chinês e o italiano, mesmo possuindo um desempenho ligeiramente inferior se comparado aos outros modelos baseados no SBERT.

Como pode ser observado trecho de código 2, foi utilizado uma integração entre o Langchain e o site Hugginface para a execução do modelo escolhido. Além disso, vale notar que a variável ”embeddings” será utilizada mais adiante no código, na etapa de criação da base de dados vetorial. Como será visto, o *framework* Langchain abstrai a complexidade da criação dos *embeddings*, bastando passar como parâmetro os *chunks* criados (em formato de texto padrão) e o modelo para transformação (selecionado neste trecho de código).

Código-fonte 2: Definição do modelo para criação dos *embeddings*

```
1 # Embeddings
```

```
2 from langchain.embeddings import HuggingFaceEmbeddings
3
4 model_name = "sentence-transformers/distiluse-base-multilingual-cased
      -v1"
5 embeddings = HuggingFaceEmbeddings(model_name=model_name)
```

Finalmente, para que seja possível armazenar grandes quantidades de informações no formato de *sentence embeddings*, é necessário utilizar bases de dados adaptadas para trabalhar com vetores multidimensionais. Essas ferramentas são chamadas de bases de dados vetoriais e são as responsáveis por permitir que o modelo a ser implementado tenha à sua disposição informações relevantes para gerar texto com base na entrada do usuário. Atualmente, existem muitas opções de bases de dados vetoriais, das quais algumas serão melhor exploradas na próxima seção.

3.4 Armazenamento dos *embeddings* em uma base de dados vetorial e busca por similaridade

Bases vetoriais são um tipo de sistema de banco de dados projetados para armazenar, gerenciar e consultar, de forma eficaz, dados vetoriais de alta dimensão. Essas ferramentas são adequadas para aplicações que se beneficiam de representações vetoriais de texto, como aprendizado de máquina, visão computacional, processamento de linguagem natural e sistemas de recomendação. Esse tipo de banco de dados diferem dos tradicionais bancos relacionais, pois são criados visando otimizar operações com dados vetoriais, na maioria dos casos, *embeddings*, enquanto os tradicionais são criados para lidar com dados muito bem estruturados.

Além disso, os bancos de dados relacionais usam *Structured Query Language* (SQL) para consultar e recuperar dados com base em critérios específicos. Essas consultas geralmente se concentram em igualdades, junções e operações de conjunto, o que favorece a utilização de dados estruturados. Por outro lado, os bancos de dados vetoriais são otimizados para realizar consultas por similaridade, permitindo a recuperação de dados com base na proximidade semântica em espaços de alta dimensão. Deste modo, esse tipo de banco utiliza cálculos de distância ou similaridade, como a distância euclidiana ou similaridade de cosseno, para permitir consultas relacionadas ao conteúdo semântico. Por exemplo, é possível buscar imagens ou documentos semelhantes com base em suas representações vetoriais.

No contexto de aplicações baseadas em RAG, as bases de dados vetoriais desempenham um papel fundamental ao aprimorar a eficiência e eficácia da etapa de recuperação de documentos. Por serem otimizados para pesquisa de similaridade e recuperação com base no conteúdo, esse tipo de banco pode encontrar e recuperar rapidamente informações, trechos ou partes de dados relevantes com base em suas representações vetoriais. Além disso, ao usar representações vetoriais, torna-se possível encontrar conteúdos semanticamente similares entre si, em vez de

depender apenas da correspondência de palavras-chave, o que leva a resultados de recuperação mais precisos e contextualmente relevantes. Outro ponto essencial para modelos RAG, é a escalabilidade, já que esses modelos podem buscar, através de grandes volumes de texto, documentos ou outras fontes de dados para fornecer informações abrangentes e diversas para gerar respostas coerentes e corretas. Por esses motivos, bancos de dados vetoriais são excelentes alternativas para armazenar a base de conhecimento externa utilizada pelos modelos.

Como citado anteriormente, a maioria das bases de dados vetoriais suporta consultas com base na similaridade de conteúdo, utilizando cálculos de distância ou similaridade. A distância Euclidiana é uma métrica comum para comparar *embeddings* de sentenças. Ela mede a distância, em uma linha reta, entre os vetores no espaço euclidiano e pode variar de zero (representação para *embeddings* iguais) até infinito, onde valores cada vez maiores representam informações cada vez menos semelhantes. No entanto, a distância Euclidiana pode ser sensível à magnitude dos vetores, o que é uma desvantagem, já que os *embeddings* podem ter diferentes magnitudes, mas ainda representarem o mesmo conceito. Assim, é importante normalizar os vetores antes de usar a distância euclidiana para compará-los ou utilizar outro método que não leve em conta as magnitudes dos vetores, como a similaridade por cosseno.

A similaridade por cosseno é frequentemente recomendada para comparar *embeddings*, especialmente em tarefas de processamento de linguagem natural e aprendizado de máquina, já que ela se concentra nas direções dos vetores e não em seu comprimento. Ademais, calcular a similaridade por cosseno é computacionalmente mais eficiente, por ser uma operação simples que envolve o produto escalar de vetores, tornando-a rápida e fácil de ser calculada, mesmo para grandes conjuntos de dados. Essa técnica reflete bem as relações semânticas entre vetores, pois vetores com ângulos próximos a 0 graus (cosseno perto de 1) estão altamente alinhados, o que sugere que os objetos representados são semelhantes em significado. Por outro lado, vetores com ângulos próximos a 90 graus (cosseno perto de 0) estão quase ortogonais, sugerindo que os objetos são distintos em significado. Por fim, como mencionado na seção de criação dos *embeddings*, o SBERT cria *sentence embeddings* otimizados para a busca por similaridade por cosseno, o que torna esse método o mais indicado para ser utilizado na implementação de um modelo baseado em RAG.

Existem diversas bases de dados vetoriais disponíveis atualmente. Elas variam desde opções de código aberto (*open-source*), como os bancos **Chroma** (CHROMADB, 2023) e **Milvus** (WANG *et al.*, 2021), opções comerciais de código fechado, como os bancos **Pinecone** (PINECONE, 2023) e **Weaviate** (WEAVIATE, 2023), e extensões que permitem bases de dados relacionais tradicionais trabalharem com dados vetorizados, o que é o caso da extensão **PGVector** (PGVECTOR, 2023) para o famoso banco **PostgreSQL** (POSTRESQL, 2023). Cada uma das opções possui suas próprias peculiaridades, que podem ser em relação à implementação de como os vetores são armazenados, às ferramentas de busca disponibilizadas por cada uma e à integração com outras plataformas, como o LangChain (CHASE, 2022).

Para a aplicação criada neste trabalho, optou-se por utilizar a extensão ”PGVector” devido à sua praticidade e à familiaridade do autor com o banco de dados PostgreSQL. Um tutorial de como instalar o banco, junto da extensão, pode ser encontrado na página do repositório”PGVector” no GitHub (PGVECTOR, 2023). Após instalado, é possível utilizá-lo na aplicação sendo desenvolvida através de sua integração com o *framework* Langchain.

A classe ”PGVecto”, disponibilizada pelo *framework*, possui vários métodos para fazer a conexão com o banco e inserção de dados. Em especial, a função ”`from_documents()`” é extremamente útil nesta etapa do desenvolvimento. Além de fazer a conexão com o banco (através do parâmetro `"connection_string"`), esta função se encarrega de transformar os *chunks* (parâmetro `"documents"`) em *embeddings*, utilizando o modelo selecionado anteriormente (parâmetro `"embedding"`). Após a transformação, os vetores criados são armazenados no banco em uma coleção com o nome definido no parâmetro ”`collection_name`”. Por fim, essa função também permite escolher a estratégia que será utilizada para a recuperação dos documentos relevantes. Como pode ser observado no trecho de código 3, a estratégia selecionada foi a similaridade por cosseno, seguindo o que foi discutido anteriormente.

Código-fonte 3: Conexão e população da base de dados utilizando o PGVector

```
1 from langchain.vectorstores.pgvector import PGVector
2 from langchain.vectorstores.pgvector import DistanceStrategy
3
4 # PGVector database
5 db = PGVector.from_documents(
6     embedding=embeddings,
7     documents=chunks,
8     collection_name="textos_brasil",
9     distance_strategy = DistanceStrategy.COSINE,
10    connection_string="postgresql://postgres:postgres@localhost:5432/
      tcc",
11 )
```

Para a recuperação de documentos, o Langchain disponibiliza a função *"similarity_search"*. Ela recebe como parâmetro uma "query", ou seja, uma consulta em formato de texto padrão. Esta consulta é automaticamente transformada em um *embedding* através do mesmo modelo indicado na etapa de população do banco. Em seguida, é aplicada a estratégia de similaridade para retornar os ”k” documentos mais similares ao *embedding* criado. É possível customizar o valor do parâmetro ”k”, alterando a quantidade de documentos recuperados. Para esta implementação, utilizou-se um $k = 5$, como pode ser observado no trecho de código 4.

Código-fonte 4: Recuperação de documentos relevantes

```
1 # Similiarity Search
2 query = "Quem descobriu o Brasil?"
```

```
3 docs = db.similarity_search(query, k=5)
```

Após recuperados, os documentos são concatenados em uma única *string*, que será utilizada como contexto para os LLMs na próxima etapa. Para realizar essa operação, é utilizado um algoritmo simples, indicado no trecho de código 5. Esse algoritmo faz, para cada um dos documentos, a extração do texto presente na variável *"page_content"*. Em seguida, os textos extraídos são concatenados em uma variável chamada `"text"`, separados por uma quebra de linha.

Código-fonte 5: Concatenação dos documentos recuperados em uma única *string*

```
1 # Relevant document concatenation
2 text = ""
3 for d in docs:
4   text += d.page_content
5   text +=  "\n"
```

O próximo e último passo para a implementação da arquitetura RAG é escolher um modelo de LLM que utilizará os dados armazenados na base de dados vetorial para gerar texto. Como será explorado na próxima seção, a escolha do modelo depende principalmente da tarefa que se deseja realizar.

3.5 Inserção dos documentos como contexto para LLMs

Atualmente, a variedade de modelos de linguagem cresce a cada dia, e muitos desses modelos são disponibilizados com apenas um pré-treinamento básico, sendo necessário fazer um ajuste fino com foco na tarefa que se deseja realizar. Dentro das tarefas que podem ser realizadas encontram-se: geração de texto, tradução, classificação, sumarização e resposta a perguntas.

No contexto da tarefa de respostas a perguntas, foco desta monografia, existem dois tipos principais: respostas extrativas e respostas abstrativas. No primeiro tipo, o LLM tem como objetivo extrair um trecho de texto de uma fonte dada que responde diretamente à pergunta do usuário. A resposta geralmente é um trecho do texto original ou ligeiramente modificado, e o modelo identifica a passagem mais relevante para gerar sua resposta. Para fazer essa identificação, o LLM avalia várias passagens candidatas e escolhe aquela que parece ter a maior probabilidade de responder corretamente à pergunta. Assim, as respostas extrativas são particularmente úteis para fornecer respostas factuais e bem estabelecidas, especialmente quando a resposta pode ser encontrada nos dados de entrada ou em uma fonte de conhecimento especificada, sendo geralmente mais conservadora e menos criativa na geração de respostas.

Já as respostas abstrativas são caracterizadas pela geração de respostas em uma forma mais próxima à linguagem humana, muitas vezes de maneira que vai além da simples extração

de texto. Nessa abordagem, o modelo tenta compreender o contexto da pergunta e gerar uma resposta usando suas próprias capacidades de geração de texto. As respostas geradas não são necessariamente trechos diretos do texto de origem, mas sim interpretações ou resumos das informações disponíveis. Dessa forma, elas são úteis para perguntas que exigem respostas interpretativas, sumarização ou uma compreensão contextual mais aprofundada. Fora isso, suas respostas podem ser mais criativas, mas também podem introduzir erros ou interpretações não intencionais.

A biblioteca "*Transfomers*"(WOLF *et al.*, 2020), para a linguagem de programação Python, disponibiliza "*pipelines*", que abstraem muito da complexidade envolvida na utilização de grandes modelos de linguagem. Cada pipeline é customizado para a tarefa que se deseja realizar. Assim, a tarefa de respostas a perguntas é realizada através do pipeline "*QuestionAnswering"*. Porém, esse pipeline só pode ser utilizado para LLMs desenvolvidos para gerar respostas extrativas. Uma vez que as abstrativas requerem um componente de interpretação e geração de texto para compor as respostas, apenas modelos disponíveis na pipeline "*Text2TextGeneration*"devem ser utilizados para essa finalidade.

No caso da tarefa de respostas a perguntas extrativas, o ajuste fino consiste em treinar o LLM utilizando um conjunto de dados, chamado de "*dataset*", composto por diversas perguntas junto de suas respectivas respostas. O *dataset* mais famoso utilizado para esse ajuste é o *Stanford Question Answering Dataset* (SQuAD) (RAJPURKAR *et al.*, 2016), um conjunto de dados contendo perguntas sobre vários artigos do site Wikipédia, onde a resposta de cada pergunta é um trecho do texto correspondente ao artigo. Além disso, em sua segunda versão, o SQuAD2.0 (RAJPURKAR; JIA; LIANG, 2018), existem algumas perguntas que não podem ser respondidas utilizando o texto fornecido, o que faz com que os modelos treinados com esse *dataset* também sejam capazes de identificar perguntas que não possuem respostas.

Neste contexto, além do ajuste fino, muitos modelos também devem ser treinados para compreender a língua portuguesa. Dois modelos capazes de entender o português são o *BERTimbau* (SOUZA; NOGUEIRA; LOTUFO, 2019), um modelo básico do BERT treinado especificamente para o português brasileiro, e o *MDeBERTa* (HE *et al.*, 2020), um modelo multilinguístico criado pela empresa Microsoft como uma melhoria dos modelos BERT e RoBERTa.

O site HugginFace (HUGGINFACE, 2023a) disponibiliza vários modelos já treinados que podem ser utilizados através da biblioteca *Transformers*. Para a implementação deste trabalho foram escolhidos dois modelos, um baseado no *BERTimbau* e outro no *MDeBERTa*, ambos treinados com o *dataset* "SQuAD". Estes modelos são o: ***"pierreguillou/bert-base-cased-squad-v1.1-portuguese"*** (HUGGINFACE, 2023b) e o ***timpal0l/mdeberta-v3-base-squad2*** (HUGGINFACE, 2023d).

Vale mencionar uma diferença fundamental entre os dois modelos sendo utilizados. O modelo baseado no *MDeBERTa* foi treinado utilizando a segunda versão do *dataset* SQuAD, em sua língua original, o inglês. É uma característica do próprio modelo ser multilinguístico e,

em função disso, ser capaz de entender as perguntas que estão sendo feitas em português. Já o *BERTimbau* é um modelo treinado inteiramente na língua portuguesa brasileira, utilizando uma tradução da primeira versão do *dataset* SQuAD, gerada através da *"Google Cloud API"*.

Analisando o trecho de código 6 é possível perceber como é facil utilizar os *pipelines* da biblioteca *Transformers*. Basta informar qual será o tipo de tarefa realizada, neste caso a tarefa *"question-answering"*, e o identificador do modelo a ser utilizado.

Código-fonte 6: Criação de *pipelines* para cada LLM

```
1 from transformers import pipeline
2
3 #MDeBERTA
4 question_answerer_MDeBERTa = pipeline("question-answering", model="
      timpal0l/mdeberta-v3-base-squad2")
5
6 #BERTimbau
7 question_answerer_BERT = pipeline("question-answering", model="
      pierreguillou/bert-base-cased-squad-v1.1-portuguese")
```

Após a criação dos *pipelines*, sua utilização também é simples. Basta passar a mesma ”query” que foi usada para recuperar os documentos relevantes (código 4) e, no parâmetro `"context"`, os próprios documentos concatenados (código 5). Desta forma, o código 7, apresentado abaixo, será utilizado para coletar a repostas dos LLMs a respeito de cinco perguntas sobre o Brasil. Todas as respostas podem ser encontradas em trechos do texto original utilizado no primeiro passo. Os resultados obtidos são apresentados na próxima seção.

Código-fonte 7: Utilização dos *pipelines* para responder perguntas

```
1 #MDeBERTA
2 q1 = question_answerer_MDeBERTa(
3     question=query,
4     context=text,
5 )
6
7 #BERTimbau
8 q2 = question_answerer_BERT(
9     question=query,
10    context=text,
11 )
12
13 print(q1)
14 print(q2)
```

3.6 Resultados Obtidos

Nesta seção será feito um comparativo entre as respostas obtidas pelos dois LLMs perante a cinco perguntas sobre o Brasil. O mesmo conjunto de documentos foi utilizado para gerar os contextos introduzidos nos dois modelos a cada pergunta. Assim, será possível avaliar se o modelo utilizado para construção dos *embeddings* permitiu a recuperação de trechos de texto relevantes para responder as perguntas. Além disso, poderá ser observado como cada modelo faz a extração da resposta presente no texto e como as diferenças em seus treinamentos afetam o desempenho obtido.

Para cada pergunta, é apresentado o contexto gerado pelos códigos 4 (recuperação dos documentos relevantes) e 5 (concatenação em uma única *string*). O trecho contendo a resposta para a pergunta foi marcado em negrito para fácil identificação. Algumas partes menos relevantes do contexto gerado foram substituídas pelo símbolo ”[...]” para diminuir o espaço ocupado. Por fim, são apresentados os resultados obtidos pelo código 7.

3.6.1 Pergunta 1: Qual a população do Brasil?

Para a pergunta **“Qual a população do Brasil?”** o contexto gerado pelos cinco documentos mais relevantes está representado abaixo.

A população do Brasil, conforme censo realizado pelo Instituto Brasileiro de Geografia e Estatística (IBGE) em 2010, foi de 190 755 799 habitantes (22,43 habitantes por quilômetro quadrado), com uma proporção de homens e mulheres de 0,96:1 e 84,36% da população definida como urbana

Brasil [...] sendo o quinto maior do mundo em área territorial (equivalente a 47,3% do território sul-americano), com 8 510 417,771 km^2, **e o sétimo em população (com 203 milhões de habitantes, em agosto de 2022)**

Os maiores aglomerados urbanos do Brasil, de acordo com a estimativa do IBGE para 2019, são as conurbações de São Paulo (com 21 656 301 habitantes), Rio de Janeiro (12 777 959), Belo Horizonte (5 178 131), Recife (4 056 323) e Brasília (4 012 896). [...]

[...] A população parda é uma categoria ampla que inclui caboclos (descendentes de brancos e indígenas), mulatos (descendentes de brancos e negros) e cafuzos (descendentes de negros e indígenas).

Em 2000, a religião católica representava 73,6% do total no país. Em 2010, era 64,6%, sendo observada pela primeira vez a redução em números absolutos (de 124 980 132 para 123 280 172). Ainda assim, o perfil religioso brasileiro continua tendo uma maioria predominantemente católica [...]

No contexto recuperado, observa-se que a resposta mais atual é "**203 milhões de habitantes**". Uma outra possível resposta para pergunta é: "**190.755.799 habitantes**", um número desatualizado há mais de 10 anos. As respostas geradas por cada modelo foram:

- BERTiumbau: “190 755 799"
- MDeBERTa: "190 755 799 habitantes"

Ambos os modelos geraram essencialmente a mesma resposta, escolhendo o trecho mais desatualizado como correto. Uma possível explicação para ambos os modelos terem escolhido a resposta mais desatualizada é o fato de que ela também é a resposta mais precisa, utilizando nove dígitos ao invés de três para descrever a quantidade de habitantes. Outra explicação é a de que a reposta extraída foi apresentada logo no primeiro documento, o mais relevante. De qualquer maneira, essa é uma indicação de que os modelos não levam em conta a ordem cronológica das possíveis respostas e buscam apenas uma informação que seja válida.

Em relação ao contexto, é possível perceber que vários trechos envolvendo o termo ”população” foram recuperados. Os dois primeiros falam sobre o número de habitantes do Brasil e contém informações para responder a pergunta. Já o terceiro apresenta o número habitantes de algumas cidades especificas, como São Paulo. Os últimos dois documentos falam sobre a descendência da população e sobre a distribuição religiosa do país. Este é forte indicativo da qualidade dos *embeddings* gerados pelo SBERT, já que todos eram relevantes para a pergunta em si.

3.6.2 Pergunta 2: Qual a justificativa da oligarquia paulista para tomar o poder em 1932?

Para a pergunta **“Qual a justificativa da oligarquia paulista para tomar o poder em 1932?”**, vários documentos falando sobre revoluções, rebeliões e trocas de poder:

> **Sob a justificativa de cobrar a implementação das promessas de reformas democráticas em uma nova constituição, em 1932 a oligarquia paulista tentou recuperar o poder através de uma revolução armada** e, [...]
>
> Com o início do governo republicano, sendo pouco mais do que uma ditadura militar, a então nova constituição de 1891 previa eleições diretas apenas para 1894 e, [...]
>
> [...], o Brasil tornou-se uma república em 1889, em razão de um golpe militar chefiado pelo marechal Deodoro da Fonseca (o primeiro presidente), [...]

> [...] Getúlio Vargas, na esteira do assassinato de João Pessoa, seu companheiro de chapa, liderar a Revolução de 1930, com o apoio dos militares, e assumir a presidência da república.
>
> No entanto, a ameaça comunista serviu de pretexto tanto para impedir as eleições previamente estipuladas, como para que Vargas e os militares lançassem mão de outro golpe de Estado em 1937 estabelecendo o Estado Novo, [...]

As repostas extraídas pelos pelos modelos foram as mesmas:

- BERTiumbau: “cobrar a implementação das promessas de reformas democráticas"
- MDeBERTa: "cobrar a implementação das promessas de reformas democráticas"

Observando as respostas, nota-se que novamente, ambos os modelos conseguiram responder a pergunta corretamente. Entretanto, ambos não extraíram o trecho completo, deixando de mencionar que a implementação das reformas se daria em "*em uma nova constituição*".

Sobre o contexto gerado, é possível observar que assim como no caso anterior, a resposta para a pergunta estava presente já no primeiro documento recuperado, confirmando que este era realmente o mais relevante. Além disso, os outros documentos recuperados, mesmo que não contendo uma relação direta com a pergunta, apresentaram informações sobre eventos onde ocorreram mudanças no poder político do país. Este é mais um indicativo da qualidade dos *sentence embeddings* criados.

3.6.3 Pergunta 3: Quais são os dois círculos imaginários que cortam o Brasil?

Para a pergunta **“Quais são os dois círculos imaginários que cortam o Brasil?”**, o contexto gerado contém o seguinte documentos:

> **O território brasileiro é cortado por dois círculos imaginários: a Linha do Equador, que passa pela embocadura do Amazonas, e o Trópico de Capricórnio, que corta o município de São Paulo**
>
> As raízes etimológicas do termo "Brasil"são de difícil reconstrução. O filólogo Adelino José da Silva Azevedo postulou que se trata de uma palavra de procedência celta [...]
>
> Atualmente o Brasil é dividido política e administrativamente em 26 estados e um distrito federal. O poder executivo é exercido por um governador e o legislativo pelas assembleias legislativas, sendo todos eleitos quadrienalmente. [...]

> [...] as estradas são as principais transportadoras de carga e de passageiros no tráfego brasileiro. O Brasil também é o sétimo mais importante país da indústria automobilística.
>
> De acordo com eles, o nome "Brasil"é derivado de "pau-brasil", designação dada a um tipo de madeira empregada na tinturaria de tecidos. [...]

As respostas geradas pelos modelos foram, novamente, exatamente as mesmas:

- BERTiumbau: "a Linha do Equador"
- MDeBERTa: "a Linha do Equador"

Nessa pergunta, ambos os modelos responderam a pergunta de forma incompleta, ao citar apenas uma das linhas imaginárias que cortam o território brasileiro. Possivelmente, os dois modelos extraíram o primeiro pedaço da resposta, nesse caso, "a Linha do Equador". Além disso, as respostas estavam separadas por um trecho explicativo, sendo ele: a passagem "que passa pela embocadura do Amazonas", que separava os trechos "Linha do Equador"e "Trópico de Capricórnio". Deste modo, as respostas geradas indicam que os modelos não são tão bons em identificar respostas com múltiplas partes, separadas por algumas palavras.

Em relação ao contexto, é possível perceber que apenas o primeiro documento oferece informações relevantes para responder a pergunta. Este é um efeito colateral da forma como os documentos são recuperados. Como a similaridade é aplicada sobre a pergunta, é natural esperar que trechos contendo a palavra "Brasil"ou que indiquem algum tipo de divisão (devido a palavra "cortam") sejam recuperados. Além disso, a função de recuperação foi configurada para retornar os cinco documentos mais similares à pergunta. Isso faz com que, mesmo que só haja um documento relevante, outros quatro documentos serão retornados. Para um humano, fica claro que os outros documentos não tem relação com as linhas imaginárias que cortam o país, mas simular essa percepção ainda é um desafio para os grandes modelos de linguagem.

3.6.4 Pergunta 4: Quais são os países que compartilham fronteiras terrestres com o Brasil?

A pergunta **“Quais são os países que compartilham fronteiras com o Brasil?”** gerou um contexto complexo, contendo dois trechos com informações suficientes para respondê-la. Ambos os trechos são compostos por diversos países e pontos cardeais em relação ao Brasil.

> [...] **compartilhando fronteiras terrestres com Uruguai ao sul; Argentina e Paraguai a sudoeste; Bolívia e Peru a oeste; Colômbia a noroeste e Venezuela, Suriname, Guiana e com o departamento ultramarino francês da Guiana Francesa ao norte.** O país

compartilha uma fronteira comum com todos os países da América do Sul, exceto Equador e Chile.

[...] faz fronteira com todos os outros países sul-americanos, exceto Chile e Equador, sendo **limitado a norte pela Venezuela, Guiana, Suriname e pelo departamento ultramarino francês da Guiana Francesa; a noroeste pela Colômbia; a oeste pela Bolívia e Peru; a sudoeste pela Argentina e Paraguai e ao sul pelo Uruguai**

O território brasileiro é cortado por dois círculos imaginários: a Linha do Equador, que passa pela embocadura do Amazonas, e o Trópico de Capricórnio, [...]

[...] é o maior país da América do Sul e da região da América Latina, sendo o quinto maior do mundo em área territorial (equivalente a 47,3% do território sul-americano), com 8 510 417,771 km², [...]

[...] O seu tamanho, relevo, clima e recursos naturais fazem do Brasil um país geograficamente diverso. O país é o quinto maior do mundo em área territorial, com 8 510 417,771 km², [...]

As respostas obtidas para cada modelo foram:

- BERTiumbau: "Equador e Chile"
- MDeBERTa: "Uruguai ao sul; Argentina e Paraguai"

Neste caso, ambos os modelos geraram respostas completamente diferentes. O BERTimbau falhou em capturar o significado semântico da palavra "exceto"e deu, como resposta, os dois únicos países da América do Sul que não compartilham fronteiras com o Brasil. Já o MDeBERTa gerou uma resposta incompleta, porém correta, citando apenas Uruguai, Argentina e Paraguai como países que fazem fronteira com o território brasileiro. Assim, particularmente nesta pergunta, nota-se uma superioridade do modelo MDeBERTa.

Em relação ao contexto, nota-se que os dois trechos que podem responder a pergunta contém essencialmente as mesmas informações, mas em ordens diferentes. Mesmo com essa diferença, a similaridade por cosseno apontou os dois trechos como os mais relevantes para gerar uma resposta. Esta é uma forte evidência de que os *sentence embeddings* conseguem capturar o sentido semântico das frases. Além disso, os outros documentos recuperados, apesar de não falarem sobre os países vizinhos ao Brasil, apresentam diversas informações sobre o território brasileiro.

3.6.5 Pergunta 5: Quem descobriu o Brasil?

O contexto gerado pelos documentos mais relevantes à pergunta **Quem descobriu o Brasil?** foi:

> Antes de ficar com a designação atual, "Brasil", as novas terras descobertas foram designadas de: Monte Pascoal (quando os portugueses avistaram terras pela primeira vez), Ilha de Vera Cruz, [...]
>
> **O território que atualmente forma o Brasil foi oficialmente descoberto pelos portugueses em 22 de abril de 1500, em expedição liderada por Pedro Álvares Cabral**. [...]
>
> De acordo com eles, o nome "Brasil"é derivado de "pau-brasil", designação dada a um tipo de madeira empregada na tinturaria de tecidos. Na época dos descobrimentos, era comum aos exploradores guardar cuidadosamente o segredo de tudo quanto achavam ou conquistavam, a fim de explorá-lo vantajosamente, [...]
>
> A música do Brasil se formou, principalmente, a partir da fusão de elementos europeus e africanos, trazidos respectivamente por colonizadores portugueses e escravos. [...]
>
> A pesquisa tecnológica no Brasil é em grande parte realizada em universidades públicas e institutos de pesquisa. [...]

Para essa última pergunta, pela primeira vez, o trecho que responde corretamente à pergunta estava presente no segundo documento mais relevante. Apesar disso, ambos os modelos conseguiram extrair corretamente a resposta ("Pedro Álvares Cabral"), mesmo quando outras respostas possíveis estavam presentes em documentos mais relevantes (como "portugueses"no primeiro documento). Isso demostra a capacidade dos LLMs de compreender contextos complexos e buscar a resposta mais correta, mesmo quando outras estão presentes no contexto.

3.6.6 Análise das resultados

Observando as respostas obtidas em cada uma das perguntas, nota-se que ambos os modelos geram respostas iguais na maior parte do tempo. Nota-se também que eles tiveram melhor desempenho quando as respostas eram compostas por trechos curtos e diretos, como é o caso das perguntas 3.6.2, e 3.6.5. Quando as respostas corretas eram formadas por trechos maiores e mais complexos, como foi o caso das perguntas 3.6.3 e 3.6.4, ambos os modelos tiveram mais dificuldade para extrair e juntar os trechos corretos para formar uma resposta coerente. Por fim, quando existe mais de um trecho que responde à pergunta corretamente, como é o caso da pergunta 3.6.1, os modelos não levaram em consideração a ordem cronológica das possíveis respostas, escolhendo apenas uma resposta que fosse válida, mesmo que desatualizada.

Outro ponto relevante a ser analisado é que ambos os modelos geraram respostas curtas e diretas. Como discutido anteriormente, esse é o objetivo da tarefa de respostas a perguntas extrativas em contraste com as abstrativas, que geram respostas mais completas. Entretanto, outro ponto que contribui para as respostas curtas é o *dataset* que foi utilizado para treinar os modelos, o SQuAD. Esse conjunto de dados é marcado pelas respostas curtas e diretas, o que faz com que os modelos respondam da mesma maneira. Para obter outros estilos de respostas, seria necessário utilizar outro conjunto de dados para treinar os modelos, abordagem que será brevemente discutida na seção 4.2 - Trabalhos Futuros.

Com relação aos contextos que foram gerados a partir dos documentos recuperados, fica claro o ótimo desempenho dos *embeddings* criados. Em todas as perguntas, o trecho contendo a resposta estava presente entre os dois textos mais relevantes recuperados. Entretanto, muitos documentos recuperados não tiveram relação direta com a pergunta sendo feita, algo fácil de se perceber por nós humanos. Este comportamento foi comentado na pergunta 3.6.3 onde apenas um documento realmente tinha relação com a pergunta, mas outros quatro foram fornecidos devido à configuração da função de recuperação. Assim, fica claro que ainda existem melhorias a serem feitas nos modelos de *embedding*, de forma que a similaridade entre os vetores seja tão precisa que se possa recuperar apenas o número necessário de documentos relevantes.

3.7 Considerações Finais

Em suma, é possível notar que existem vários passos envolvidos na customização de um modelo de linguagem através da arquitetura RAG. Cada passo compreende diversas abordagens que devem ser comparadas e estudadas na hora de implementar o modelo, como por exemplo a estratégia para a divisão dos documentos, os modelos de *embeddings* disponíveis, as diferentes bases de dados vetoriais existentes e os diversos LLMs desenvolvidos para cada tarefa em específico.

Fica clara também a qualidade dos vetores criados a partir do SBERT. Para cada pergunta, a similaridade por cosseno foi aplicada em todos os **242** *embeddings* em apenas alguns segundos. Isto é uma melhora de desempenho incrível se comparada aos outros métodos de recuperação de documentos relevantes.

Por fim, é possível observar que muito trabalho ainda pode ser feito em relação à tarefa de respostas a perguntas, em especial no contexto da língua portuguesa, para melhorar as respostas extraídas pelos modelos aqui descritos. Em perguntas com respostas simples e curtas, os dois modelos estudados conseguem entender o contexto e extrair a resposta corretamente. Entretanto, para perguntas com respostas mais complexas, os modelos muitas vezes geram respostas incompletas, desatualizadas ou até mesmo incorretas.

Capítulo 4

CONCLUSÃO

4.1 Contribuições

A principal contribuição desta monografia foi a discussão dos passos necessários para a implementação da arquitetura RAG, juntamente dos conceitos e ferramentas relevantes para cada etapa. Primeiramente, foram estudados diferentes modos de definir o tamanho ideal para cada *chunk* criado a partir dos documentos da base de conhecimento externa, juntamente de três técnicas para a realização do processo de divisão.

Em seguida, foram detalhadas algumas das abordagens em relação a criação de *embeddings*, em especial a utilidade e, muitas vezes, a necessidade da utilização de *sentence embeddings* para a implementação de modelos baseados em RAG. Não só isso, também foi apresentada a ferramenta SBERT, capaz de criar representações vetoriais utilizando uma versão aprimorada do famoso modelo de linguagem BERT.

Logo após, foram destacadas as vantagens da utilização de uma base de dados vetorial para o armazenamento e a recuperação dos *embeddings*. Além de apresentar algumas das soluções disponíveis no mercado, fez-se um breve comparativo entre dois métodos para a recuperação de documentos: a distância euclidiana e a semelhança por cosseno.

Depois, comentou-se a grande variedade de modelos de linguagem disponíveis atualmente, dando ênfase na tarefa de respostas a perguntas. Mostrou-se que existem dois tipos de respostas que podem ser geradas pelos modelos em questão: as extrativas e as abstrativas. Além disso, foi estudada, brevemente, a biblioteca Transformers, a qual disponibiliza *"pipelines"* para simplificar a utilização dos LLMs. Explorou-se também o processo de ajuste fino destes modelos, utilizando *datasets* personalizados para cada tarefa, citando o "SQuAD"como o *dataset* mais utilizado para o treinamento de modelos voltados a respostas extrativas. Ademais, foram citados dois LLMs ajustados para trabalharem com a língua portuguesa brasileira, o BERTimbau e o MDeBERTa.

Finalmente, a breve implementação realizada, com foco na tarefa de resposta a perguntas, serviu para exemplificar uma das possíveis utilizações da arquitetura RAG. A aplicação criada também demonstrou a qualidade dos *embeddings* gerados através do SBERT por meio dos contextos gerados para cada pergunta. Além disso,a análise dos resultados evidenciou que ainda há espaço para melhorias nos modelos de linguagem utilizados, principalmente em relação à

língua portuguesa.

Em termos profissionais, essa monografia e todo o processo de pesquisa envolvido em sua escrita, trouxe ao autor uma base de conhecimento pouco explorada pelas matérias da graduação. A etapa de implementação da arquitetura também proporcionou um aprendizado mais aprofundado de como diferentes tecnologias se conectam para criar uma aplicação baseada em grandes modelos de linguagem. Deste modo, essa monografia proporcionou uma valiosa contribuição para a formação do autor como engenheiro de computação, especialmente em áreas modernas como inteligência artificial, processamento de linguagem natural e bases de dados vetoriais.

4.2 Limitações e Trabalhos Futuros

Neste trabalho, foram exploradas apenas algumas das opções disponíveis para a implementação de cada etapa da arquitetura RAG. Vale ressaltar que este assunto ainda está se desenvolvendo e criando maturidade, visto que o artigo descrevendo essa arquitetura foi publicado em 2020, apenas três anos atrás (LEWIS *et al.*, 2020). Com isso, trabalhos futuros podem atualizar os conceitos explorados nesta monografia, assim como adicionar novas ferramentas e métodos para a realização de cada etapa da implementação. Outrossim, a implementação utilizada como exemplo neste trabalho foi focada na tarefa de respostas extrativas. Trabalhos futuros permitem a exploração de como modelos de linguagem desenvolvidos para outras tarefas, como sumarização e tradução de textos, podem se beneficiar da utilização da arquitetura RAG.

No contexto da tarefa de respostas extrativas, a análise das respostas obtidas deixou evidente que melhorias ainda podem ser feitas para aumentar o desempenho dos modelos. Em especial, destacou-se que as respostas curtas e diretas eram decorrentes do *dataset* "SQuAD", utilizado para o treinamento de ambos os modelos utilizados. Outros conjuntos de dados, com respostas mais longas, podem ser aplicados na etapa de ajuste fino para gerar respostas mais completas. Um famoso modelo desenvolvido para essa finalidade foi o "ELI5", criado pelo grupo de pesquisa do Facebook (FAN *et al.*, 2019). Esse *dataset* compreendia uma série de postagens advindas do fórum *"Explain Like I'm Five"* (explique-me como se tivesse cinco anos, em português) do site Reddit, onde uma comunidade online fornecia respostas, muitas vezes longas e detalhadas, para as mais diversas perguntas. Entretanto, em abril de 2023, o site Reddit mudou sua política de APIs, tornando a fonte dos dados desse *dataset* inacessível. Deste modo, trabalhos futuros podem explorar a utilização de outros conjuntos de dados para o ajuste fino dos modelos de linguagem, em busca de respostas mais longas e completas.

4.3 Considerações Sobre o Curso de Graduação

O curso de graduação em Engenharia de Computação, oferecido pela EESC e ICMC, possui um nível de qualidade elevado. Sua grade curricular, que abrange uma grande variedade de assuntos, e o nível elevado dos conteúdos estudados e das avaliações, forçam os alunos deste curso a desenvolverem uma capacidade de aprendizado diferenciada. O ponto forte do curso é ser capaz de formar engenheiros capazes de se adaptar e aprender rapidamente conceitos e tecnologias novas, habilidade que se prova um diferencial no mercado de trabalho.

Porém, essa habilidade só é adquirida com muito esforço, dedicação e sacrifício. O elevado nível de exigência do curso muitas vezes impede o aluno de Engenharia de Computação de se desenvolver em outras áreas como esportes, extracurriculares e projetos pessoais. Em muitas disciplinas, existe uma discrepância gritante entre os créditos informados na plataforma JupiterWeb e o que realmente é cobrado na sala de aula. É o caso de disciplinas como SEL0621 Projetos de Circuitos Integrados Digitais I, SEL0612 Ondas Eletromagnéticas e SEL0618 Projetos de Circuitos Integrados Analógicos, que não possuem nenhum crédito trabalho alocado, mas consomem preciosas horas, fora do horário de aula, com trabalhos e relatórios extensos. Dessa forma, o curso de Engenharia de Computação poderia se beneficiar muito de uma reestruturação de créditos ou de uma cobrança mais rigorosa para que professores sigam o programa proposto e os créditos alocados para cada disciplina.

REFERÊNCIAS

BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A.; AGARWAL, S.; HERBERT-VOSS, A.; KRUEGER, G.; HENIGHAN, T.; CHILD, R.; RAMESH, A.; ZIEGLER, D. M.; WU, J.; WINTER, C.; HESSE, C.; CHEN, M.; SIGLER, E.; LITWIN, M.; GRAY, S.; CHESS, B.; CLARK, J.; BERNER, C.; MCCANDLISH, S.; RADFORD, A.; SUTSKEVER, I.; AMODEI, D. Language models are few-shot learners. **CoRR**, abs/2005.14165, 2020. Disponível em: <https://arxiv.org/abs/2005.14165>. Citado na página 17.

CHASE, H. **LangChain**. 2022. Disponível em: <https://github.com/langchain-ai/langchain>. Acesso em: 01/11/2023. Citado 3 vezes nas páginas 21, 23 e 35.

CHROMADB. **ChromaDB**. 2023. Disponível em: <https://docs.trychroma.com/>. Acesso em: 01/11/2023. Citado 2 vezes nas páginas 18 e 35.

DEVLIN, J.; CHANG, M.; LEE, K.; TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. **CoRR**, abs/1810.04805, 2018. Disponível em: <http://arxiv.org/abs/1810.04805>. Citado na página 22.

FAN, A.; JERNITE, Y.; PEREZ, E.; GRANGIER, D.; WESTON, J.; AULI, M. ELI5: long form question answering. **CoRR**, abs/1907.09190, 2019. Disponível em: <http://arxiv.org/abs/1907.09190>. Citado na página 48.

HE, P.; LIU, X.; GAO, J.; CHEN, W. Deberta: Decoding-enhanced BERT with disentangled attention. **CoRR**, abs/2006.03654, 2020. Disponível em: <https://arxiv.org/abs/2006.03654>. Citado 2 vezes nas páginas 23 e 38.

HUGGINFACE. **Huggin Face models**. 2023. Disponível em: <https://huggingface.co/models>. Acesso em: 01/11/2023. Citado 2 vezes nas páginas 23 e 38.

____. **pierreguillou/bert-base-cased-squad-v1.1-portuguese**. 2023. Disponível em: <https://huggingface.co/pierreguillou/bert-base-cased-squad-v1.1-portuguese>. Acesso em: 01/11/2023. Citado na página 38.

____. **sentence-transformers/distiluse-base-multilingual-cased-v1**. 2023. Disponível em: <https://huggingface.co/sentence-transformers/distiluse-base-multilingual-cased-v1>. Acesso em: 01/11/2023. Citado na página 33.

____. **timpal0l/mdeberta-v3-base-squad2**. 2023. Disponível em: <https://huggingface.co/timpal0l/mdeberta-v3-base-squad2>. Acesso em: 01/11/2023. Citado na página 38.

KIROS, R.; ZHU, Y.; SALAKHUTDINOV, R.; ZEMEL, R. S.; TORRALBA, A.; URTASUN, R.; FIDLER, S. Skip-thought vectors. **CoRR**, abs/1506.06726, 2015. Disponível em: <http://arxiv.org/abs/1506.06726>. Citado na página 32.

LEWIS, P. S. H.; PEREZ, E.; PIKTUS, A.; PETRONI, F.; KARPUKHIN, V.; GOYAL, N.; KÜTTLER, H.; LEWIS, M.; YIH, W.; ROCKTÄSCHEL, T.; RIEDEL, S.; KIELA, D. Retrieval-augmented generation for knowledge-intensive NLP tasks. **CoRR**, abs/2005.11401, 2020. Disponível em: <https://arxiv.org/abs/2005.11401>. Citado 3 vezes nas páginas 18, 24 e 48.

LIU, Y.; OTT, M.; GOYAL, N.; DU, J.; JOSHI, M.; CHEN, D.; LEVY, O.; LEWIS, M.; ZETTLEMOYER, L.; STOYANOV, V. Roberta: A robustly optimized BERT pretraining approach. **CoRR**, abs/1907.11692, 2019. Disponível em: <http://arxiv.org/abs/1907.11692>. Citado na página 22.

MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. Efficient estimation of word representations in vector space. **CoRR**, abs/1301.3781, 2013. Disponível em: <https://arxiv.org/abs/1301.3781>. Citado na página 32.

MUSSATTO, M. **mmussatto/Wikipedia-Brasil-Page github repository**. 2023. Disponível em: <https://github.com/mmussatto/Wikipedia-Brasil-Page>. Acesso em: 01/11/2023. Citado na página 30.

OPENAI. **ChatGPT**. 2023. Disponível em: <https://chat.openai.com>. Acesso em: 01/11/2023. Citado na página 17.

PGVECTOR. **Open-source vector similarity search for Postgres**. 2023. Disponível em: <https://github.com/pgvector/pgvector>. Acesso em: 01/11/2023. Citado 3 vezes nas páginas 18, 35 e 36.

PICHAI, S. **An important next step on our AI journey**. 2023. Disponível em: <https://blog.google/technology/ai/bard-google-ai-search-updates/>. Acesso em: 01/11/2023. Citado na página 17.

PINECONE. **The Pinecone Vector Database**. 2023. Disponível em: <https://www.pinecone.io/product/>. Acesso em: 01/11/2023. Citado na página 35.

POSTRESQL. **PostgreSQL: The World's Most Advanced Open Source Relational Database**. 2023. Disponível em: <https://www.postgresql.org/>. Acesso em: 01/11/2023. Citado 2 vezes nas páginas 18 e 35.

RAJPURKAR, P.; JIA, R.; LIANG, P. Know what you don't know: Unanswerable questions for squad. **CoRR**, abs/1806.03822, 2018. Disponível em: <http://arxiv.org/abs/1806.03822>. Citado na página 38.

RAJPURKAR, P.; ZHANG, J.; LOPYREV, K.; LIANG, P. Squad: 100, 000+ questions for machine comprehension of text. **CoRR**, abs/1606.05250, 2016. Disponível em: <http://arxiv.org/abs/1606.05250>. Citado na página 38.

REIMERS, N.; GUREVYCH, I. Sentence-bert: Sentence embeddings using siamese bert-networks. **CoRR**, abs/1908.10084, 2019. Disponível em: <http://arxiv.org/abs/1908.10084>. Citado na página 32.

SBERT.NET. **SBERT Pretrained Models**. 2023. Disponível em: <https://www.sbert.net/docs/pretrained_models.html>. Acesso em: 01/11/2023. Citado na página 33.

SCHWABER-COHEN, R. **Chunking Strategies for LLM Applications**. 2023. Disponível em: <https://www.pinecone.io/learn/chunking-strategies/>. Acesso em: 01/11/2023. Citado na página 29.

SOUZA, F.; NOGUEIRA, R.; LOTUFO, R. Portuguese named entity recognition using bert-crf. **arXiv preprint arXiv:1909.10649**, 2019. Disponível em: <http://arxiv.org/abs/1909.10649>. Citado na página 38.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. **CoRR**, abs/1409.3215, 2014. Disponível em: <http://arxiv.org/abs/1409.3215>. Citado na página 18.

TOPSAKAL, O.; AKINCI, T. C. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. **International Conference on Applied Engineering and Natural Sciences**, v. 1, n. 1, p. 1050–1056, Jul. 2023. Disponível em: <https://as-proceeding.com/index.php/icaens/article/view/1127>. Citado na página 24.

TOUVRON, H.; LAVRIL, T.; IZACARD, G.; MARTINET, X.; LACHAUX, M.-A.; LACROIX, T.; ROZIèRE, B.; GOYAL, N.; HAMBRO, E.; AZHAR, F.; RODRIGUEZ, A.; JOULIN, A.; GRAVE, E.; LAMPLE, G. **LLaMA: Open and Efficient Foundation Language Models**. 2023. Citado na página 17.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L.; POLOSUKHIN, I. Attention is all you need. **CoRR**, abs/1706.03762, 2017. Disponível em: <http://arxiv.org/abs/1706.03762>. Citado na página 21.

WANG, J.; YI, X.; GUO, R.; JIN, H.; XU, P.; LI, S.; WANG, X.; GUO, X.; LI, C.; XU, X.; YU, K.; YUAN, Y.; ZOU, Y.; LONG, J.; CAI, Y.; LI, Z.; ZHANG, Z.; MO, Y.; GU, J.; JIANG, R.; WEI, Y.; XIE, C. Milvus: A purpose-built vector data management system. In: **Proceedings of the 2021 International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGMOD '21), p. 2614–2627. ISBN 9781450383431. Disponível em: <https://doi.org/10.1145/3448016.3457550>. Citado na página 35.

WEAVIATE. **The AI Native Vector Database**. 2023. Disponível em: <https://weaviate.io/>. Acesso em: 01/11/2023. Citado na página 35.

WIKIPÉDIA. **Brasil**. 2023. Disponível em: <https://pt.wikipedia.org/wiki/Brasil>. Acesso em: 01/11/2023. Citado 2 vezes nas páginas 19 e 27.

WOLF, T.; DEBUT, L.; SANH, V.; CHAUMOND, J.; DELANGUE, C.; MOI, A.; CISTAC, P.; RAULT, T.; LOUF, R.; FUNTOWICZ, M.; DAVISON, J.; SHLEIFER, S.; PLATEN, P. von; MA, C.; JERNITE, Y.; PLU, J.; XU, C.; SCAO, T. L.; GUGGER, S.; DRAME, M.; LHOEST, Q.; RUSH, A. M. Transformers: State-of-the-art natural language processing. In: **Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations**. Online: Association for Computational Linguistics, 2020. p. 38–45. Disponível em: <https://www.aclweb.org/anthology/2020.emnlp-demos.6>. Citado 3 vezes nas páginas 21, 23 e 38.