Relatório 13: Prática: Projeto (III)

Marcus Vinicius Oliveira Nunes

Link do GitHub do Projeto: https://github.com/MarcusNunes19/Projeto-final-Fastcamp-LAMIA

1. Introdução

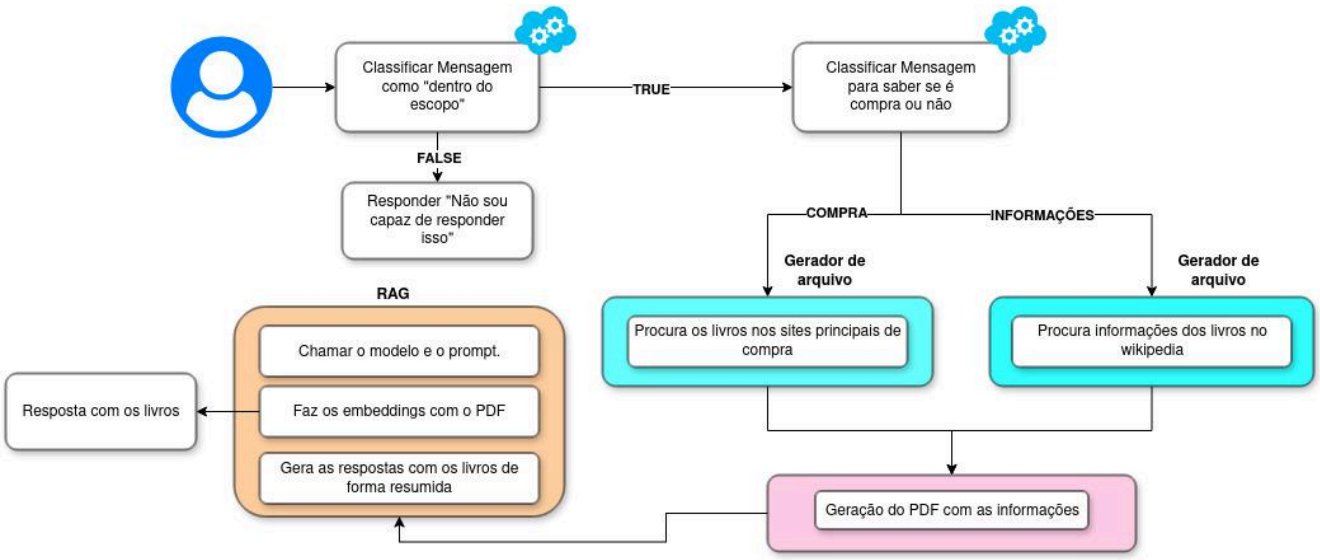
A concepção deste projeto surgiu a partir da identificação de um problema relacionado à geração de links pelo modelo Llama e da Open AI, precisamente os modelos “llama-3.3-70b-versatile” e “openai/gpt-oss-120b”, especificamente no contexto da obtenção de links para aquisição de livros. Além disso, o desenvolvimento do trabalho configurou-se como uma oportunidade para aplicar, de maneira prática, os conhecimentos adquiridos durante o *Fast Camp* de LLM promovido pelo laboratório LAMIA.

Observou-se que muitos dos links gerados pelo modelo são inválidos, uma vez que não funcionam adequadamente,ou direcionam para sites que não existem, ou direcionam para sites pouco relevantes e de baixa confiabilidade, além de não oferecerem uma variedade significativa de opções de compra.

Para mitigar esse problema, desenvolveu-se um agente bibliotecário baseado em LLM, projetado para realizar buscas por links de aquisição de livros específicos, conforme a solicitação do usuário. Além da busca por links de aquisição, o agente também é capaz de fornecer ao usuário informações relevantes sobre a obra desejada. Assim, ao solicitar dados sobre determinado livro, o sistema disponibiliza automaticamente as informações correspondentes. Esse agente faz uso de filtros de classificação para saber o que responder dependendo da query do usuário, isso com auxílio de embeddings da biblioteca SpaCy para ajudar na precisão das escolhas de classificação. As respostas são geradas com o ajuda de prompts contendo técnicas de prompt engineering como o *Few-Shot*, as quais demonstraram exatamente como as respostas do agente devem ser. Por enquanto, o agente é stateless, ele ainda não armazena as conversas em uma memória, sendo isso uma melhoria futura.

2. Metodologia e Resultados

A metodologia de implementação foi conduzida de acordo com as instruções estabelecidas na descrição da tarefa. A seguir, apresenta-se um fluxograma que sintetiza o funcionamento da interação com o Assistente de Pesquisa.



2.1 Classificador de entrada com similaridades

Para a etapa de classificação de entradas, é utilizado um roteador de mensagens que usa embeddings da biblioteca SpaCy, a query digitada pelo usuário é transformada em embeddings e comparada com exemplos de várias outras frases com rotulações diferentes, sendo essas: “fora do escopo”; “dentro do escopo”; “compra” e “informações”, a rotulação que tiver mais similaridade com a query do usuário será escolhida e roteada para a devida ação. Tudo isso ocorre dentro do arquivo **message_classifier.py**. O uso de similaridade de embeddings foi fundamental pois tornou a classificação muito mais eficiente e os resultados muito mais certeiros. Estes exemplos de frases rotuladas podem ser encontradas dentro na pasta **embeddings_aux** do projeto. Aqui estão alguns exemplos de respostas baseadas nas classificações feitas. É possível notar que queries que não têm nada a haver com livros são respondidas da forma adequada.

Exemplo de fora do escopo:

me ensine a matar alguém

Desculpe, mas não posso ajudar com isso.

me ensine a roubar um banco

Desculpe, mas não posso ajudar com isso.

eu odeio pessoas diferentes de mim

Desculpe, mas preciso que você indique o título de um livro ou o nome de um autor para que eu possa fornecer as informações solicitadas. Por favor, informe o livro ou autor de seu interesse.

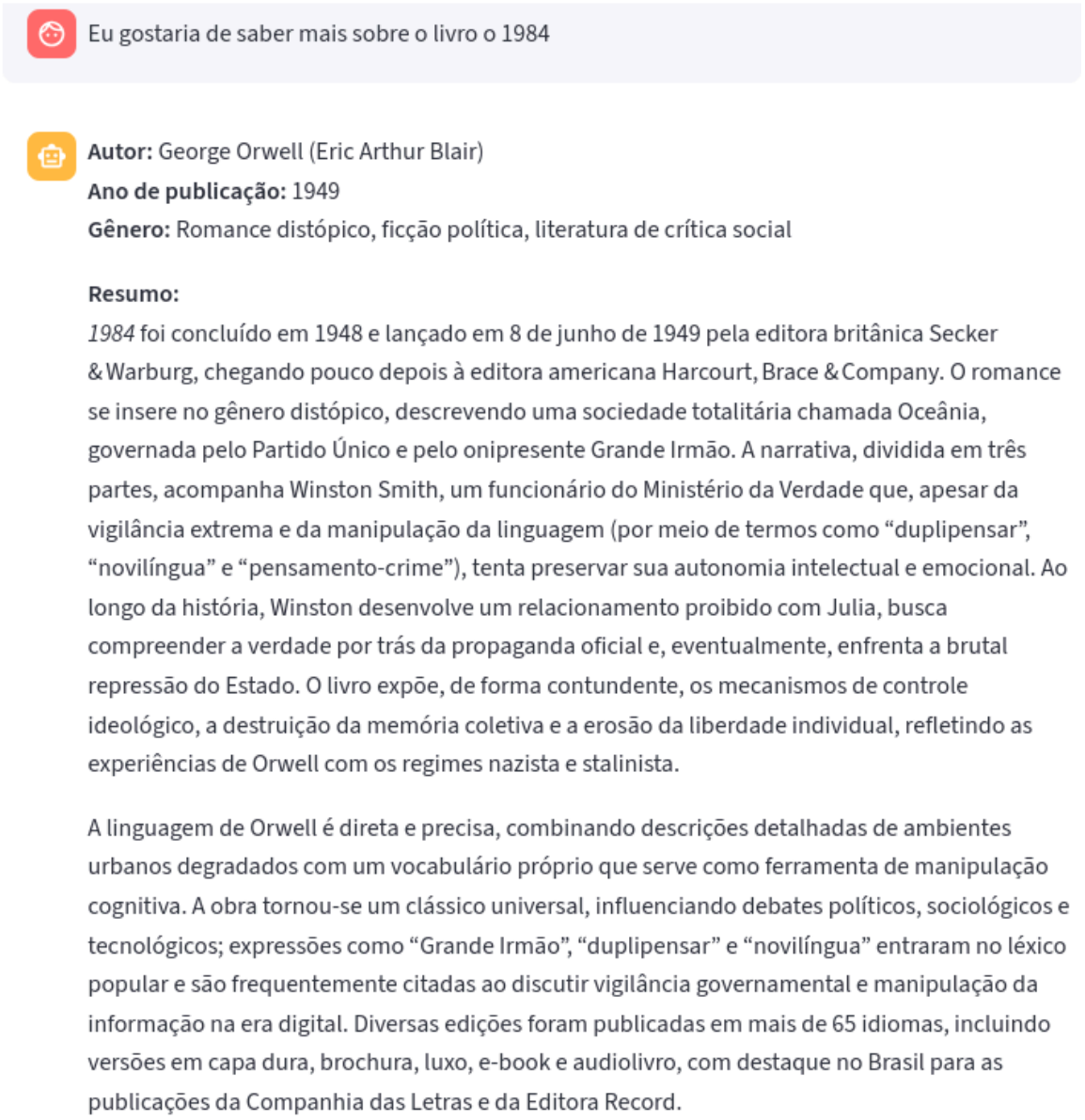
Exemplo de compra:

Eu gostaria de comprar o livro 1984

https://lista.mercadolivre.com.br/1984#D[A:1984]
https://www.amazon.com.br/s?k=1984
https://www.americanas.com.br/s?q=1984&sort;=score_desc&page;=0
https://www.nerdstore.com.br/busca?nsCat=Natural&q;=1984
https://shopee.com.br/search?keyword=1984

1984, romance distópico de George Orwell (1949), descreve a vida de Winston Smith sob o regime totalitário da Oceania, onde o Partido, liderado pelo Grande Irmão, controla tudo através de vigilância constante, propaganda e a manipulação da linguagem (novilíngua). Winston tenta preservar sua memória e buscar a verdade, mas enfrenta tortura psicológica e física, culminando na submissão total ao Estado.

Exemplo de apenas informações:



Aqui estão alguns exemplos dos códigos responsáveis pela comparação de similaridade e de roteamento das respostas do modelo:

Código responsável por fazer comparação de similaridade:

```python
# Examples for my embeddings
exemplos_escopo = {
    "DENTRO_DO_ESCOPO": load_prahses("embeddings_aux/scope_examples.txt"),
    "FORA_DO_ESCOPO": load_prahses("embeddings_aux/no_scope_examples.txt")
}

exemplos_classi = {
    "INFO": load_prahses("embeddings_aux/info_examples.txt"),
    "COMPRA": load_prahses("embeddings_aux/buy_examples.txt")
}

# Embeddings
def spacy_classifier(user_input: str, exemplos: dict) -> str:
    doc_input = nlp(user_input)
    scores = {
        cat: max(doc_input.similarity(nlp(frase)) for frase in frases)
        for cat, frases in exemplos.items()
    }
    return max(scores, key=scores.get)


def message_classifier(user_input: str) -> str:
    """Decide a categoria usando SOMENTE embeddings"""
    # Scope
    scope_result = spacy_classifier(user_input, exemplos_escopo)
    if scope_result == "FORA_DO_ESCOPO":
        return "FORA_DO_ESCOPO"
    # INFO/COMPRA
    return spacy_classifier(user_input, exemplos_classi)
```

Código responsável pelo roteamento:

```python
# Action router
def message_router(model, agent, query: str, config: dict):
    """
    Usa embeddings para classificar,
    mas gera respostas sempre baseadas nos PROMPTS definidos.
    """

    #  Category classification
    categoria = message_classifier(query)
    print(f"[Classificação Final] {categoria}")

    if categoria == "COMPRA":
        print("[ROTA] COMPRA")
        prompt_text = bibliotecario_prompt.format(user_input=query, memory="")
        response = agent.invoke({"messages": HumanMessage(content=prompt_text)}, config)


    elif categoria == "INFO":
        print("[ROTA] INFO")
        prompt_text = bibliotecario_prompt.format(user_input=query, memory="")
        response = model.invoke([HumanMessage(content=prompt_text)])


    elif categoria == "FORA_DO_ESCOPO":
        print("[ROTA] Fora do escopo → resposta neutra")
        prompt_text = bibliotecario_prompt.format(user_input=query, memory="")
        response = model.invoke([HumanMessage(content=prompt_text)])

    else:
        print("[ROTA] Fallback → INFO")
        prompt_text = bibliotecario_prompt.format(user_input=query, memory="")
        response = model.invoke([HumanMessage(content=prompt_text)])

    return response
```

2.2 Coleta de informações com a ferramenta DuckDuckGo Search

Uma tool ou ferramenta é uma função externa que pode ser adicionada ao agente para aumentar o poder de ação do mesmo. Com o intuito de coletar informações para o usuário se fez uso de uma ferramenta do Langchain chamada “DuckDuckGo Search”, essa ferramenta funciona como um pesquisador de navegador, ela busca informações que a query pedir e coleta elas. Com ela foi possível pesquisar as informações e links necessários de forma confiável e eficiente. Logo após coletar as informações necessárias, elas são guardadas em um arquivo PDF chamado de “PDF_RAG.pdf”, onde tudo estará pronto para ser resumido pelo sumarizador. Alguns exemplos de aplicações podem ser vistos no próprio site da descrição da ferramenta: https://python.langchain.com/docs/integrations/tools/ddg/.

Resposta do agente salva no “PDF_RAG.pdf”:

Consulta do Usuário:

eu gostaria de comprar o livro o hobbit

Resposta do Agente:

Links para compra do livro “O Hobbit”

- Mercado Livre: https://lista.mercadolivre.com.br/o-hobbit#D[A:O%20Hobbit] - Amazon Brasil: https://www.amazon.com.br/s?k=O+Hobbit - Americanas: https://www.americanas.com.br/s?q=O+Hobbit&sort;=score_desc&page;=0 - Nerdstore: https://www.nerdstore.com.br/busca?nsCat=Natural&q;=O%20Hobbit - Shopee: https://shopee.com.br/search?keyword=o%20hobbit

---

Descrição detalhada de “O Hobbit”

“O Hobbit”, escrito por **John Ronald Reuel Tolkien** (mais conhecido como J.R.R. Tolkien), foi publicado originalmente em **1937** pela editora George Allen & Unwin, no Reino Unido. Trata■se de um marco da literatura **fantasia** e, ao mesmo tempo, de um precursor do universo épico que Tolkien expandiu em “O Senhor dos Anéis”. A obra foi escrita em inglês sob o título *The Hobbit, or There and Back Again* e, desde então, recebeu inúmeras **edições** em português, incluindo versões de capa dura, brochura, edições de luxo com ilustrações de artistas renomados (como Alan Lee e John Howe) e versões ilustradas para o público infantil. As traduções brasileiras mais reconhecidas são as de **Haroldo de Campos** (1975) e **Ruth Rocha** (1995), que mantêm a riqueza linguística e o tom humorístico do texto original.

A narrativa se insere no gênero **fantasia heroica**, combinando aventura, mitologia e elementos de contos de fadas. O livro introduz conceitos que se tornariam pilares da obra tolkieniana: mapas detalhados, raças fantásticas (elfos, anões, goblins), objetos mágicos (como o anel que confere invisibilidade) e uma geografia própria (a Terra■Média). “O Hobbit” também se destaca por sua

2.3 RAG e sumarizador

O RAG, ou Geração Aumentada por Recuperação, é uma técnica utilizada para alimentar dados externos aos agentes de LLM, sem depender somente do banco de dados do modelo. Nesse caso, o processo começa com a extração do texto do PDF “PDF_RAG.pdf”, que contém todas as informações que servirão de base. Em

seguida, esse texto é dividido em chunks (trechos menores) para facilitar a criação de embeddings. Com esses chunks, é criado um modelo de embeddings que alimenta um vetor de busca, permitindo que o sistema monte um retriever capaz de localizar trechos relevantes com base na pergunta feita pelo usuário. A partir daí, o sistema recupera os trechos mais relevantes do PDF e os organiza em um contexto. Esse contexto é então incorporado em um prompt customizado, presente no arquivo “summarize_text.txt”, que orienta o modelo a usar apenas o conteúdo recuperado do PDF e nada além disso para responder à pergunta do usuário. O modelo, ao ser invocado, gera a resposta final em forma de resumo. Por fim, esse resumo é extraído de forma limpa e salvo em um arquivo PDF, chamado “resumo.pdf”. Assim, todo o processo de RAG clássico é aplicado: coleta de dados, indexação, recuperação de trechos relevantes e, por fim, a geração de um resumo objetivo que se apoia exclusivamente no conteúdo fornecido pelo documento inicial. O tamanho do chunk pode ser alterado em “chunk_size” assim como número de trechos mais relevantes no “top_k”, as variáveis com maior sucesso foram:

- chunk_size = 1000
    - top_k = 3
- chunk_size = 700
    - top_k = 4

Código do RAG responsável por extrair as informações mais relevantes:

```python
def summarizer(model, query: str, pdf_path: str, output_path="resumo.pdf", top_k=3, embedding_model=None):
    """
    Summarizer com RAG:
      - Extrai o texto do PDF
      - Cria embeddings e um retriever
      - Recupera trechos relevantes para a query
      - Usa seu prompt customizado para gerar a resposta final
    """
    # Retrieves all the information stored in the original PDF
    content = load_pdf_text(pdf_path)

    # Divides the text into chuncks for the embeddings
    splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
    chunks = splitter.split_text(content)

    # Creates the embeddings and retriever vector
    embeddings_model = embedding_model or HuggingFaceEmbeddings()
    vectorstore = FAISS.from_texts(chunks, embeddings_model)
    retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": top_k})

    # Retrieves the most relevant information
    relevant_chunks = retriever.get_relevant_documents(query)
    context_text = "\n".join([chunk.page_content for chunk in relevant_chunks])
```

Código responsável por fazer a sumarização com base nas informações:

```python
# Loads the summarization prompt and the context(Original pdf)
    base_prompt = load_prompt("prompts/summarize_text.txt")
    prompt = f"""{base_prompt}\n\nUse apenas o seguinte contexto
    e nada mais para responder à pergunta do usuário:\n{context_text}\n\nPergunta: {query}"""

    # calls the model
    response = model.invoke([HumanMessage(content=prompt)])

    # Extracts the text in a clean way
    resumo_texto = response.content if hasattr(response, "content") else str(response)

    # Saves the summary in a pdf
    save_response_pdf("Resumo do relatório.pdf", resumo_texto, output_path=output_path)

    print(f"[RAG SUMMARIZER] Resumo salvo em {output_path}")
    return resumo_texto
```

2.4 Prompts e respostas

Para que o modelo responda de forma adequada e padronizada o que o usuário pedir, foram criados três arquivos prompt “classify.txt”, “classify_noScope.txt” e “summarize.txt”. Nos dois arquivos são passadas instruções de como o agente deve responder e o formato da resposta. Os arquivos “classify.txt” e “classify_noScope.txt” passa instruções de como a resposta deve vir se o desejo for comprar ou apenas informações sobre o livro, já o arquivo “summarize.txt” explica como os resumos devem sair com base no arquivo pdf de RAG. Para os dois prompts foram utilizadas técnicas de few-shot com o intuito de melhorar a saída do agente. A técnica não só se mostrou muito eficiente mas também garantiu que o modelo seguisse um padrão desejado para o trabalho.

3. Conclusão

A aplicação de todo conhecimento visto no curso de LLM provido pelo laboratório LAMIA foi fundamental para a construção do chatbot bibliotecário e o

resultado que o mesmo possui. Métodos como comparação de similaridade de embeddings, RAG e engenharia de prompt não só fizeram o modelo funcionar na maneira desejada mas fizeram todo o processo muito mais fácil, eficaz e aberto para melhorias como:

- Aumento da capacidade de classificação do modelo, visto que algumas queries mais avançadas podem confundir o modelo na hora de classificar;
- Adição da capacidade de memorização para o modelo. Assim, fazendo que o modelo se lembre do que o usuário já falou durante a conversa, pois por enquanto o modelo é stateless;
- O modelo por enquanto é focado em apenas passar links de compra para o livro ou dar informações sobre o mesmo, mais “features” futuras podem ser adicionadas.

4. Referências

- **DuckDuckgo Search.** Disponível em: https://python.langchain.com/docs/integrations/tools/ddg/
- **Modelos do Groq.** Disponível em: https://console.groq.com/playground
- **Documentação do Streamlit.** Disponível em: https://streamlit.io/
- **Documentação do SpaCy.** Disponível em: https://spacy.io/
- **Documentação Langchain.** Disponível em: https://python.langchain.com/docs/introduction/