**Documentação da Fase 2: Pipeline de Ingestão de Dados (RAG)**

**Visão Geral da Fase 2**

A Fase 2 do nosso projeto teve como objetivo principal construir a **Base de Conhecimento** para o nosso sistema RAG (Retrieval Augmented Generation). Isso envolveu a transformação de dados brutos (arquivos CSV de tickets) em informações estruturadas e pesquisáveis para um Large Language Model (LLM) hospedado localmente via Ollama.

**As principais etapas executadas foram:**

1. **Leitura e Pré-processamento:** Carregar os dados dos arquivos CSV, gerar IDs únicos para cada ticket e formatá-los em documentos textuais coesos.
2. **Fragmentação (Chunking):** Dividir esses documentos em pedaços menores (chunks) de tamanho gerenciável, com sobreposição para manter o contexto.
3. **Geração de Embeddings:** Converter cada chunk em um vetor numérico (embedding) usando um modelo de embedding local (Ollama com nomic-embed-text).
4. **Armazenamento no Banco de Dados Vetorial:** Inserir os chunks e seus embeddings em um banco de dados vetorial (ChromaDB) para permitir buscas eficientes por similaridade semântica.

**Desafios Enfrentados e Soluções**

Durante a implementação desta fase, encontramos alguns desafios importantes, principalmente relacionados à comunicação entre os componentes (Python, Ollama, ChromaDB) e ao gerenciamento de recursos.

**Desafio 1: Conexão Recusada ao Servidor Ollama (ConnectionRefusedError)**

* **Sintoma:** O script Python apresentava o erro ConnectionRefusedError: [WinError 10061] Nenhuma conexão pôde ser feita porque a máquina de destino as recusou ativamente ao tentar gerar embeddings. Isso impedia o script de progredir na ingestão de dados.
* **Causa Raiz:** A causa principal era que o servidor Ollama não estava ativo e rodando na porta esperada (11434), ou uma instância anterior do Ollama (ou outro processo) estava ocupando essa porta, impedindo o servidor de iniciar corretamente. O erro bind: Normalmente é permitida apenas uma utilização de cada endereço de soquete confirmou que a porta estava em uso.
* **Solução Implementada:**
  1. **Verificação e Encerramento de Processos:** Usamos comandos como netstat -ano | findstr :11434 para identificar o PID do processo que estava utilizando a porta 11434 e taskkill /PID [SEU\_PID] /F para encerrá-lo.
  2. **Garantia de Início do Ollama Server:** Foi reiterada a necessidade de iniciar o servidor Ollama em um terminal **separado** usando ollama serve (ou ollama run [modelo]) e mantê-lo ativo durante a execução do script Python.
  3. **Teste de Conexão com Ollama:** Implementamos um pequeno teste no script Python para verificar a comunicação com o modelo de embedding do Ollama antes de iniciar a ingestão em massa, garantindo que o servidor estivesse responsivo.

**Desafio 2: Ingestão de Dados Massivos Aparentemente Travada e Lenta**

* **Sintoma:** Após a correção do erro de conexão, o script Python parecia "travar" na etapa de "Armazenando chunks e embeddings no ChromaDB", com a pasta chroma\_db não mostrando crescimento de tamanho por longos períodos.
* **Causa Raiz:** O processo de geração de embeddings para 20.000 documentos (que se traduzem em muitos milhares de chunks) é computacionalmente intensivo. Cada chunk exige uma chamada ao modelo de embedding do Ollama. O processo, embora funcional, era demorado e não fornecia feedback granular de progresso.
* **Solução Implementada:**
  1. **Limpeza do Ambiente:** Recomendou-se a exclusão da pasta chroma\_db (rmdir /s /q chroma\_db) antes de cada nova execução de ingestão para garantir um início limpo.
  2. **Redução para Teste Rápido:** Adicionamos uma linha ao script (documentos\_processados\_dicts = documentos\_processados\_dicts[:100]) para permitir o processamento de apenas 100 documentos, acelerando as iterações de teste e confirmando que o pipeline funcionava em pequena escala.
  3. **Monitoramento:** Orientamos a verificação do uso de CPU/RAM (via Gerenciador de Tarefas) e dos logs do terminal do Ollama para confirmar que os processos estavam ativos e processando, mesmo que lentamente.

**Desafio 3: Dessincronização de Versões do Script**

* **Sintoma:** O usuário não conseguia encontrar as linhas comentadas para teste no script local, apesar de eu as ter mencionado, causando confusão.
* **Causa Raiz:** Houve um erro de minha parte ao fornecer uma versão incompleta do script anteriormente, onde as linhas para depuração e teste não estavam realmente inseridas no código que o usuário havia copiado.
* **Solução Implementada:** Forneci a versão **completa e corrigida** do script ingestao\_dados.py, garantindo que todas as funcionalidades e comentários de depuração estivessem presentes, permitindo que o usuário ativasse as linhas de teste conforme o planejado.

**Código Final da Fase 2 (ingestao\_dados.py)**

Este é o script ingestao\_dados.py finalizado da Fase 2, que realiza a leitura dos CSVs, o pré-processamento, o chunking, a geração de embeddings e o armazenamento no ChromaDB.

Python

# ingestao\_dados.py

import pandas as pd

import os

import uuid

# --- Importações LangChain e ChromaDB ---

from langchain\_community.document\_loaders import DataFrameLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.embeddings import OllamaEmbeddings

from langchain\_community.vectorstores import Chroma

from langchain.docstore.document import Document # Importar Document explicitly

# --- Função de Carregamento e Formatação ---

def carregar\_e\_formatar\_documentos(diretorio\_csv: str, colunas\_conteudo: list) -> list:

"""

Carrega arquivos CSV de um diretório, gera um ID único para cada linha

e formata o conteúdo de cada linha em um documento de texto.

Args:

diretorio\_csv (str): Caminho para o diretório contendo os arquivos CSV.

colunas\_conteudo (list): Lista de nomes das colunas cujo conteúdo deve ser

incluído no texto do documento.

Returns:

list: Uma lista de dicionários, onde cada dicionário representa um documento

formatado com seu ID e conteúdo. Ex:

[{'id': '...', 'content': '...'}, {'id': '...', 'content': '...'}]

"""

documentos\_formatados = []

id\_contador = 0

print(f"Lendo arquivos CSV de: {diretorio\_csv}")

for nome\_arquivo in os.listdir(diretorio\_csv):

if nome\_arquivo.endswith('.csv'):

caminho\_completo = os.path.join(diretorio\_csv, nome\_arquivo)

print(f"Processando arquivo: {nome\_arquivo}")

try:

try:

df = pd.read\_csv(caminho\_completo, encoding='utf-8')

except UnicodeDecodeError:

df = pd.read\_csv(caminho\_completo, encoding='latin1')

except Exception as e:

print(f"Erro de encoding ao ler {nome\_arquivo}: {e}. Tentando sem especificar encoding.")

df = pd.read\_csv(caminho\_completo)

colunas\_validas = [col for col in colunas\_conteudo if col in df.columns]

if not colunas\_validas:

print(f"Aviso: Nenhuma coluna de conteúdo válida encontrada em {nome\_arquivo}. Pulando.")

continue

for index, row in df.iterrows():

id\_contador += 1

document\_id = f"doc\_{id\_contador}" # Ou str(uuid.uuid4())

texto\_documento = f"## Documento ID: {document\_id}\n"

for col\_name in colunas\_validas:

value = row.get(col\_name)

if pd.notna(value):

texto\_documento += f"\*\*{col\_name.replace('\_', ' ').title()}\*\*: {value}\n"

texto\_documento += "\n"

documentos\_formatados.append(

{'id': document\_id, 'content': texto\_documento, 'source': nome\_arquivo}

)

except Exception as e:

print(f"Erro ao ler ou processar {nome\_arquivo}: {e}")

return documentos\_formatados

# --- FUNÇÃO PRINCIPAL ---

if \_\_name\_\_ == "\_\_main\_\_":

# --- CONFIGURAÇÃO ---

DIRETORIO\_DOS\_CSVS = 'data' # Ajuste para o nome da sua pasta de dados

NOME\_DO\_ARQUIVO\_CSV = 'tickets.csv' # Nome específico do arquivo dentro da pasta

COLUNAS\_PARA\_CONTEUDO = ['subject', 'body', 'answer', 'type', 'queue', 'priority', 'language', 'tag\_1', 'tag\_2', 'tag\_3', 'tag\_4', 'tag\_5', 'tag\_6', 'tag\_7', 'tag\_8']

# --- CONFIGURAÇÕES DE CHUNKING E EMBEDDING ---

CHUNK\_SIZE = 1000 # Tamanho máximo de caracteres por chunk

CHUNK\_OVERLAP = 200 # Quantidade de caracteres que se sobrepõem entre chunks

EMBEDDING\_MODEL = "nomic-embed-text" # Modelo de embedding a ser usado com Ollama

# --- CONFIGURAÇÃO DO CHROMADB ---

# Caminho onde o ChromaDB irá armazenar os dados. Será uma pasta dentro do seu projeto.

PERSIST\_DIRECTORY = "./chroma\_db"

# --- EXECUÇÃO ---

if not os.path.exists(DIRETORIO\_DOS\_CSVS):

print(f"Erro: O diretório '{DIRETORIO\_DOS\_CSVS}' não foi encontrado.")

print("Por favor, crie esta pasta e coloque seus arquivos CSV dentro dela (ex: 'tickets.csv').")

else:

# Carrega e formata os documentos brutos

documentos\_processados\_dicts = carregar\_e\_formatar\_documentos(

DIRETORIO\_DOS\_CSVS,

COLUNAS\_PARA\_CONTEUDO

)

if not documentos\_processados\_dicts:

print("Nenhum documento foi processado. Verifique os arquivos CSV e as configurações.")

else:

print(f"\nTotal de {len(documentos\_processados\_dicts)} documentos brutos processados.")

### NOVIDADE: REDUZINDO DOCUMENTOS PARA TESTE (DESCOMENTE PARA TESTAR RAPIDAMENTE) ###

# documentos\_processados\_dicts = documentos\_processados\_dicts[:100]

# print(f"DEBUG: Limitando a {len(documentos\_processados\_dicts)} documentos para teste rápido.")

langchain\_documents = []

for doc\_dict in documentos\_processados\_dicts:

langchain\_documents.append(

Document(page\_content=doc\_dict['content'], metadata={'id': doc\_dict['id'], 'source': doc\_dict['source']})

)

print(f"Convertidos {len(langchain\_documents)} documentos brutos para formato LangChain Document.")

# --- 2. Fragmentar (Chunking) os Documentos ---

print(f"Dividindo documentos em chunks (tamanho: {CHUNK\_SIZE}, overlap: {CHUNK\_OVERLAP})...")

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=CHUNK\_SIZE,

chunk\_overlap=CHUNK\_OVERLAP,

length\_function=len,

add\_start\_index=True

)

chunks = text\_splitter.split\_documents(langchain\_documents)

print(f"Total de chunks gerados: {len(chunks)}")

print("\nExemplo do primeiro chunk gerado:\n")

print("--- INÍCIO DO EXEMPLO DO CHUNK ---")

print(chunks[0].page\_content)

print(chunks[0].metadata)

print("--- FIM DO EXEMPLO DO CHUNK ---")

# --- 3. Gerar Embeddings com Ollama ---

print(f"\nInicializando modelo de embedding '{EMBEDDING\_MODEL}' com Ollama...")

try:

embeddings\_model = OllamaEmbeddings(model=EMBEDDING\_MODEL)

### NOVIDADE: TESTE DE EMBEDDING ANTES DA INGESTÃO EM MASSA ###

print("Realizando um teste rápido de embedding...")

test\_text = "Isso é um texto de teste para o embedding."

embedding\_test = embeddings\_model.embed\_query(test\_text)

print(f"Teste de embedding bem-sucedido. Tamanho do vetor: {len(embedding\_test)} dimensões.")

print("Continuando com a ingestão em massa...")

except Exception as e:

print(f"ERRO CRÍTICO: Falha ao inicializar/testar o modelo de embedding Ollama: {e}")

print("A ingestão não pode continuar. Verifique:")

print("1. Se o Ollama Server está rodando no seu terminal (ex: 'ollama serve').")

print("2. Se o modelo 'nomic-embed-text' foi baixado corretamente ('ollama pull nomic-embed-text').")

print("3. Se o Ollama tem recursos de memória suficientes.")

exit()

# --- 4. Armazenar no ChromaDB ---

print(f"\nArmazenando {len(chunks)} chunks e embeddings no ChromaDB em '{PERSIST\_DIRECTORY}'...")

try:

vectorstore = Chroma.from\_documents(

documents=chunks,

embedding=embeddings\_model,

persist\_directory=PERSIST\_DIRECTORY

)

vectorstore.persist()

print("\nDados armazenados com sucesso no ChromaDB.")

print(f"Número de documentos no ChromaDB: {vectorstore.\_collection.count()}")

print(f"ChromaDB persistido em: {os.path.abspath(PERSIST\_DIRECTORY)}")

except Exception as e:

print(f"ERRO CRÍTICO: Falha ao armazenar dados no ChromaDB: {e}")

print("Verifique se há permissões de escrita no diretório e se o ChromaDB está configurado corretamente.")

print("\nFASE 2 (Injestão de Dados) CONCLUÍDA COM SUCESSO!")