Inteligência Artificial 2

Aula 19: O RAG Corporativo (Spring AI na Prática)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

Onde Paramos?

Na aula anterior, fizemos o **Hello World do Spring AI**:

- Spring Initializr + dependência `spring-ai-openai`.
- `ChatClient` injetado automaticamente pelo Spring.
- Endpoint `/api/pergunta` respondendo com IA generativa.

A Pergunta de Hoje

O endpoint responde com conhecimento **geral**.
Como conectá-lo aos **documentos da empresa** via RAG?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Modelar** o pipeline RAG como ETL: Extract → Transform → Load.
2. **Implementar** `DocumentReader`, `TokenTextSplitter` e `VectorStore`.
3. **Configurar** o `QuestionAnswerAdvisor` para interceptar prompts automaticamente.
4. **Explicar** o padrão Adapter que desacopla o banco vetorial do código.
5. **Testar** o RAG com perguntas dentro e fora do domínio.

O Amadurecimento do RAG

- Na Aula 17, operamos a Inteligência Artificial no modo exploratório. Escrevemos a matemática do *Retrieval-Augmented Generation* (RAG) do zero em *scripts* Python.
- No cenário corporativo real (bancos, seguradoras, e-commerce), esse script isolado não sobrevive. Aplicações exigem alta disponibilidade, multithreading, telemetria, segurança *OAuth2* e arquiteturas de banco de dados segregadas.
- A estrutura procedural de arquivos locais cede lugar ao **Design Orientado a Objetos** e **Arquiteturas Hexagonais** suportadas pelo ecossistema *Spring Boot*.

Scripts Monolíticos vs. Arquiteturas de Camadas

- Um *script Jupyter* une extração de texto, comunicação com API da OpenAI e lógica de negócios num fluxo único. (Forte acoplamento).
- O **Spring AI** aplica injeção de dependência e responsabilidade única (SOLID).
- A leitura de documentos, a fragmentação semântica, a representação vetorial (*Embedding*) e o processamento cognitivo (*ChatClient*) passam a ser interfaces distintas, interligáveis como blocos de montar (*Lego*).

O Pipeline RAG como um ETL

A indexação de dados para um banco de IA é essencialmente um *Pipeline* **ETL (Extract, Transform, Load)** clássico da engenharia de dados.

- **E (Extract):** Resgatar o texto bruto de PDFs, bases SQL ou buckets S3.
- **T (Transform):** Limpar o texto, particionar (*chunking*) e calcular o vetor matemático (*Embedding*).
- **L (Load):** Inserir os tensores no Banco de Dados Vetorial.

O Spring AI oferece abstrações perfeitas para cada uma destas etapas.

Spring AI: A Interface `Document`

- No ecossistema Java, um texto não trafega no ETL como uma *String* solta. Ele é envelopado na classe `Document`.
- A classe possui dois atributos centrais:
  1. `content` (String): O conteúdo textual literal do fragmento.
  2. `metadata` (Map¡String, Object¿): Dicionário chave-valor guardando dados vitais como *"autor"*, *"nome_arquivo"*, *"numero_pagina"*.
- Os metadados são cruciais para a filtragem (*Metadata Filtering*) e para citar fontes no *frontend*.

Fase de Extração: `DocumentReader`

- Interface que extrai o texto de fontes de dados heterogêneas e entrega uma lista de instâncias `Document` preenchidas.
- **Implementações Padrão:** `TikaDocumentReader` (lê Word, PDF, Excel usando a biblioteca Apache Tika), `PagePdfDocumentReader` (lê PDFs preservando delimitação de páginas).
- A abstração permite que amanhã você passe a ler manuais de uma API REST sem que a etapa seguinte de transformação perceba a diferença.

Fase de Transformação: `DocumentTransformer`

Documentos extensos extrapolam o *Context Window* dos LLMs e causam "diluição" semântica do *Embedding* (o vetor final perde a especificidade do tema).

- O `DocumentTransformer` atua na lista de `Document` gerada pelo leitor.
- A implementação mais vital é o `TokenTextSplitter`. Ele avalia algoritmicamente onde fatiar o documento.
- Não fragmenta cortando palavras no meio. Quebra na pontuação mais próxima para respeitar o limite de *N Tokens* configurado.

A Matemática do TokenTextSplitter

- Exemplo prático de corte: Tamanho de *chunk* = 500 *Tokens*, com *Overlap* = 100 *Tokens*.
- O *Overlap* (sobreposição) injeta 100 tokens do final do fragmento 1 no início do fragmento 2. Isso evita a destruição de sentido se o algoritmo cortar o texto exatamente no clímax da argumentação.

```
// Transformacao Declarativa
var splitter = new TokenTextSplitter(500, 100, 5, 10000, true);
List<Document> chunks = splitter.apply(documentosBrutos);
```

Fase de Carga: `DocumentWriter`

- Interface destinada a salvar a lista de `Document` resultante em um coletor de armazenamento definitivo.
- Em pipelines de inteligência artificial, o `DocumentWriter` final invariavelmente aponta para o Banco de Dados Vetorial.

Abstração de Bancos: A Interface `VectorStore`

- O Spring AI provê a interface unificadora `VectorStore`. Você desenvolve a consulta através do método `vectorStore.similaritySearch(String query)`.
- O banco real que fará a busca de Vizinhos Mais Próximos (ANN, HNSW) é definido por injeção de dependência.
- **Provedores compatíveis:** ChromaDB, Pinecone, Qdrant, Milvus, Redis, Neo4j, e PostgreSQL (`pgvector`).

Padrão de Projeto Adapter no Spring AI

A flexibilidade de trocar o banco vetorial instantaneamente é garantida pelo Padrão de Projeto **Adapter** (GoF).

- Quando adicionamos a dependência `spring-ai-pgvector-store`, o Spring injeta um adaptador `PgVectorStore` que implementa a interface `VectorStore`.
- Esse adaptador *traduz* a chamada genérica de similaridade do Java para uma *query* SQL com a sintaxe proprietária (*e.g.*, `SELECT * FROM items ORDER BY embedding <-> '[...]' LIMIT 5`).
- O código cliente permanece agnóstico ao SGBD subjacente, minimizando o débito técnico.

O Motor de Vetorização (`EmbeddingModel`)

- Uma dúvida clássica: *"Onde está o código que chama o modelo matemático para transformar o Chunk num vetor de `float`?"*
- O Spring Boot encapsula essa complexidade. Ao instanciar um `VectorStore`, o framework exige a injeção conjunta de um `EmbeddingModel`.
- Quando enviamos um arquivo para ser salvo, o próprio Banco Vetorial (no adaptador do Spring) invoca a API do Google ou OpenAI (*EmbeddingModel*), gera o tensor denso e o insere no banco, tudo de forma transparente.

Implementando Ingestão de PDF em Java

```
Componentpublic class ExtratorDocumental private final VectorStore vectorStore; //
    Adaptador injetado do bancopublic ExtratorDocumental(VectorStore vectorStore)
    this.vectorStore = vectorStore;public void processarPdf(Resource arquivoPdf)
    var reader = new PagePdfDocumentReader(arquivoPdf);var splitter = new
    TokenTextSplitter();// Le, fragmenta, vetoriza e indexa em 1
    linhathis.vectorStore.accept(splitter.apply(reader.get()));
```

O Desafio da Fase de Geração

Como vimos em Python (Aula 17), após recuperar o vetor similar, tínhamos de concatenar manualmente strings gigantescas do *System Prompt*.

- Isso acarreta alto risco de *Prompt Injection* e deixa o controlador engessado e sujo.
- O Spring AI resolveu essa dor introduzindo o conceito poderoso da Programação Orientada a Aspectos (AOP) através dos **Advisors**.

A Mágica do `QuestionAnswerAdvisor`

O `QuestionAnswerAdvisor` é um interceptador padrão do `ChatClient`.

- Quando um usuário requisita a rota do controlador (*"Diferencie A de B"*), esse *Advisor* intercepta a chamada antes dela atingir a OpenAI.
- Ele extrai a pergunta, executa o `similaritySearch` autonomamente no `VectorStore`, recupera os `Documents` e **reescreve dinamicamente** as mensagens do *System* injetando as restrições arquiteturais ("Responda com base nisto...").
- O código cliente Java reduz-se a meras 3 linhas focadas no negócio.

O Controlador REST RAG (Na Prática)

```
RestControllerRequestMapping("/api/chat")
public class RagController {
    private final ChatClient chatClient;

    public RagController(ChatClient.Builder builder, VectorStore
        vectorStore) {
        this.chatClient = builder
            .defaultAdvisors(new QuestionAnswerAdvisor(vectorStore))
            .build();
    }

    GetMapping("/perguntar")public String
        responderDuvida(RequestParam String q) {
        return chatClient.prompt().user(q).call().content();
    }
}
```

Testes Unitários de RAG (Mocking)

- Em arquiteturas Enterprise, nenhuma rota vai para Produção sem Testes de Integração e Unidade.
- Devido às puras abstrações em interfaces, testar o seu controlador de RAG dispensa invocar GPUs reais e gastar dólares em requisições de teste.
- O programador usa bibliotecas (como Mockito) para fornecer um `VectorStore` *Mock* e um `ChatModel` sintético local (ex: `SimpleVectorStore`), validando a lógica do microsserviço no CI/CD (Github Actions) em poucos milissegundos.

Múltiplos `Advisors` (Pipeline de Contexto)

Além do RAG puro, o desenvolvedor pode "empilhar" interceptadores.

- `MessageChatMemoryAdvisor`: Intercepta requisições e salva o histórico das respostas numa tabela relacional (ou Redis), garantindo que a API *Stateless* consiga dialogar como se tivesse memória do que foi falado nas rotas HTTP anteriores.
- `SafeGuardAdvisor`: Filtra a entrada por palavrões e detecta intenção maliciosa antes de despachar a carga sensível para o banco vetorial.

Conclusão da Aula 19

- Migrar o RAG para uma aplicação Spring AI Java transforma provas de conceito acadêmicas em sistemas industriais altamente manuteníveis.
- O padrão *Extract*, *Transform*, *Load (ETL)* padronizou a conversão ruidosa de PDFs e textos num motor vetorial escalável.
- A genialidade arquitetural dos **Advisors** livrou a Engenharia de Software da manipulação confusa de matrizes de *strings*, integrando o raciocínio semântico de Inteligência Artificial numa sintaxe coesa que times consolidados de *Backend* conhecem e confiam.

Dúvidas?

Email: alessio@cefetmg.br
Aula: Segunda e Quarta-Feira, 7h as 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Lembrete

Confira sua Issue no GitLab!

Próxima aula: RAG Corporativo

100% da Aula 19 Concluída!