Inteligência Artificial 2

Aula 17: RAG Parte 1 (A Magia Feita à Mão em Python)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

1. O Limite dos Modelos Generativos
2. O Paradigma RAG
3. Pipeline de Ingestão (Indexação)
4. Pipeline de Consulta e Recuperação
5. A Geração Aumentada
6. Implementação em Python

Onde Paramos?

Na aula anterior, dominamos a **Engenharia de Prompts Sistêmica**:

- System Prompts que controlam persona e restrições do LLM.
- JSON Mode para saída estruturada e determinística.
- Temperatura baixa ($T = 0.1$) para extração precisa de dados.

A Pergunta de Hoje

O LLM sabe muito, mas **alucina** sobre dados internos da empresa.
Como injetar **documentos reais** no prompt?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Explicar** o problema da alucinação e por que o RAG resolve.
2. **Implementar** o pipeline Retrieve $\rightarrow$ Augment $\rightarrow$ Generate do zero em Python.
3. **Conectar** a um banco ChromaDB e fazer busca por similaridade.
4. **Construir** um prompt rígido que ancora o LLM nos dados locais.
5. **Testar** guardrails: perguntas fora do domínio devem ser recusadas.

O Limite da Memória Paramétrica

- Durante a fase de *Pre-training*, um LLM comprime trilhões de *tokens* da internet nos seus bilhões de pesos sinápticos (matrizes).
- Esta base de conhecimento internalizada é chamada de **Memória Paramétrica**. Ela é estática, ou seja, o modelo não aprende fatos novos após o término do treinamento (cujo custo chega a dezenas de milhões de dólares).
- A compressão é *lossy* (com perda). O modelo aprende relações semânticas e gramaticais com perfeição, mas memoriza dados factuais de maneira probabilística e imperfeita.

A Anatomia da Alucinação

O que é a Alucinação na IA Generativa?

É o fenômeno onde o modelo gera textos que são gramaticalmente perfeitos, fluentes e extremamente convincentes, mas factualmente incorretos, ilógicos ou infundados.

- Ocorre porque o objetivo matemático do *Decoder* é **maximizar a verossimilhança** (*likelihood*) da próxima palavra. Ele foi otimizado para soar plausível, não para validar a veracidade.
- Quando um dado não existe na *Memória Paramétrica*, a probabilidade das palavras "chutadas" é amostrada, gerando invenções coerentes.

O Dilema Corporativo: Dados Privados

Como aplicar LLMs em casos de uso corporativo?

- Uma empresa deseja um assistente que responda perguntas baseadas nos seus manuais de RH, planilhas financeiras restritas ou contratos de clientes.
- Esses documentos **nunca** fizeram parte do conjunto de treinamento da OpenAI ou do Google.
- Solução ingênua: Colocar todo o contrato no *Prompt*.
- O Problema: Janelas de Contexto (*Context Windows*) possuem limites (historicamente de $4K$ a $128K$ tokens), e enviar 10.000 páginas de manuais a cada requisição geraria latência impraticável e custos inviáveis (*Input Tokens*).

A Ilusão do Fine-Tuning para Fatos

- **Mito Comum:** "Vamos fazer um *Fine-Tuning* no Llama-3 com os PDFs da nossa empresa para ele aprender nossos processos."
- **A Realidade:** *Fine-Tuning* altera a distribuição de probabilidade superficial. É excelente para ensinar a IA a adotar um novo **formato** (ex: falar como Shakespeare ou formatar XML), mas é desastroso para injetar fatos precisos.
- Um modelo ajustado sofre de Esquecimento Catastrófico (*Catastrophic Forgetting*) e as alucinações persistem, agora mascaradas pelo tom corporativo da empresa.

A Solução: Retrieval-Augmented Generation (RAG)

Para solucionar o dilema factual, o artigo de Lewis et al. (2020) formalizou o paradigma **RAG (Geração Aumentada por Recuperação)**.

- A filosofia do RAG inverte a lógica da arquitetura. Em vez do modelo recorrer à sua memória paramétrica confusa, transferimos o papel da *verdade* para um banco de dados externo infalível.
- O LLM é relegado puramente a ser um **motor de processamento de linguagem**, não mais uma base de conhecimento.

O Paradigma da Separação de Responsabilidades

No RAG, temos duas camadas ortogonais de memória:

- **Memória Não-Paramétrica (Externa):** O repositório real de documentos (PDFs, Wikis, Bancos SQL). É 100% acurada, facilmente atualizável e barata para armazenar.
- **Memória Paramétrica (O LLM):** Sabe *como* ler, *como* traduzir, *como* resumir e *como* extrair, mas é proibido de usar o próprio conhecimento para gerar respostas sobre o domínio privado.

O Fluxo de Ingestão (Index Pipeline)

O RAG não é um processo mágico de 1 passo; é composto de dois *pipelines* de engenharia de dados separados. O primeiro é a Ingestão (Preparação do Conhecimento).

1. **Carregamento (Loading):** Scripts lêem PDFs, DOCXs ou páginas HTML.
2. **Fragmentação (Chunking):** O texto é dividido em pedaços menores e controláveis.
3. **Vetorização (Embedding):** Cada fragmento é passado num modelo especial para ser transformado em tensor geométrico.
4. **Armazenamento (Indexing):** Os tensores e os metadados são salvos no *Vector Database*.

Fragmentação de Texto (Chunking)

Um documento de 1.000 páginas não pode ser vetorizado como um objeto único. Ele sofrerá da maldição da dimensionalidade (diluição semântica).

- **Fixed-Size Chunking:** Quebrar o texto a cada 500 *tokens* ou 1.000 caracteres, com uma "zona de sobreposição" (*Overlap*) de 100 caracteres para evitar quebrar uma frase no meio.
- **Semantic Chunking:** Quebrar respeitando hierarquia de Markdown (Headers, Sections) ou quebras de parágrafo naturais (`\n\n`).

Cada "chunk" representará um vetor individual no banco de dados.

O Motor de Representação (Embedding Model)

- A etapa vital do processo de ingestão é o algoritmo de **Text Embedding** (visto detalhadamente em aulas passadas).
- Um modelo de Embedding (como `text-embedding-3-small` da OpenAI ou os abertos da BAAI) recebe o *chunk* de texto e devolve um vetor denso $v \in \mathbb{R}^d$ (ex: $d = 1536$).
- Parágrafos que discutem "Multas Contratuais" terão vetores matematicamente próximos (alta similaridade de cosseno) a parágrafos sobre "Quebra de Contrato".

Bancos de Dados Vetoriais (Vector Databases)

Os vetores de dezenas de milhares de *chunks* são armazenados num banco especializado para IA: o **Vector Database** (Pinecone, ChromaDB, Milvus, Qdrant ou pgvector).

- Bancos relacionais normais não conseguem buscar "qual array de float é mais parecido com esse outro array?".
- Bancos Vetoriais usam algoritmos *Approximate Nearest Neighbor (ANN)* como o **HNSW (Hierarchical Navigable Small World)** para encontrar vizinhos vetoriais mais próximos em escala de bilhões de entradas em milissegundos.

O Fluxo de Consulta (Retrieval Pipeline)

Com o Banco Vetorial populado, o sistema passa a operar o *pipeline* em tempo real para o usuário final:

1. **Pergunta:** O usuário final digita uma *query*.
2. **Vetorização da Pergunta:** A pergunta sofre transformação de Embedding.
3. **Busca Semântica:** O sistema busca os K trechos mais próximos.
4. **Síntese (Augmentation):** O sistema cria o super-prompt.
5. **Geração (Generation):** O LLM responde.

Métrica de Similaridade Semântica

Durante a busca (Retrieval), o *backend* recebe o vetor da pergunta Q e compara com os milhões de vetores do banco V. A métrica mais utilizada é a **Similaridade do Cosseno**:

$$\text{Cosine Similarity}(Q, V) = \frac{Q \cdot V}{\|Q\|_2 \|V\|_2} \tag{1}$$

O cálculo reflete o ângulo entre o vetor da pergunta e o vetor do documento, independente da magnitude (tamanho) de cada texto. Um valor próximo de 1 garante que o conceito semântico do documento responde a pergunta.

Recuperação (Top-K Retrieval)

- A consulta retornará, baseada na métrica de distância, os **Top-K** fragmentos.
- K é um hiperparâmetro arquitetural vital. Geralmente $K = 3$ ou $K = 5$.
- Nós queremos puxar os 5 parágrafos do documento corporativo que melhor descrevem a dúvida do usuário e extraí-los puramente como Texto (*String*) da memória do banco para injetar no Prompt.

A Geração Aumentada (Augmentation)

Esta é a fase de Engenharia de Prompts do RAG. O desenvolvedor escreve uma *System Instruction* implacável.

O Condicionamento do Contexto

A instrução força que o modelo de Linguagem enxergue apenas o contexto acoplado. *"O seu mundo se resume ao texto a seguir. Qualquer conhecimento prévio deve ser ignorado."*

Estruturação do Prompt RAG

```
# Exemplo simplificado de concatenacao
prompt_aumentado = f"""
Voce e o assistente de RH da empresa.
Responda a pergunta do usuario utilizando APENAS o contexto abaixo.
Se a resposta nao estiver no contexto, diga: "Nao possuo esta
    informacao".

CONTEXTO DO BANCO DE DADOS VETORIAL:
{texto_recuperado_1}
{texto_recuperado_2}

PERGUNTA DO USUARIO:
{pergunta_usuario}
"""
```

A Morte da Alucinação (Garantia Grounded)

- Um sistema RAG perfeitamente configurado é **Grounded** (Ancorado ou Fundamentado).
- Porque todo o texto gerado decorre de um *chunk* específico do banco, é trivial na engenharia de software extrair os metadados do *chunk* e exibir "Citações" ou "Fontes" na interface web.
- O usuário clica na Citação "1" e a tela exibe o PDF exato e a página onde a IA fundamentou a resposta, instaurando auditoria e verificação (*Verifiability*) absoluta.

Código: Inicializando ChromaDB em Python

```python
import chromadb
from chromadb.utils import embedding_functions

# Inicializa o VectorDB local (Em memoria ou Persistente)
client = chromadb.PersistentClient(path="./meu_banco_vetorial")

# Inicializa a funcao de embedding (ex: OpenAI, Default, etc)
embed_fn = embedding_functions.DefaultEmbeddingFunction()

# Cria a colecao (Tabela vetorial)
collection = client.get_or_create_collection(name="contratos",
    embedding_function=embed_fn)

# Inserindo os Chunks
collection.add(
    documents=["A multa por rescisão é de 20%", "Férias devem ser
        avisadas com 30 dias"],
    metadatas=[{"fonte": "RH", "pagina": 2}, {"fonte": "RH", "pagina
```

Código: O RAG Manual Completo

```python
import google.generativeai as genai

pergunta = "Qual é o valor da multa?"

# Etapa 1: Retrieval (Top-K=1)
resultados = collection.query(query_texts=[pergunta], n_results=1)
contexto_str = resultados["documents"][0][0] # Acessando o texto
    retornado

# Etapa 2: Augmentation
prompt = f"Baseado APENAS nisto:\n{contexto_str}\nResponda: {
    pergunta}"

# Etapa 3: Generation
model = genai.GenerativeModel("gemini-1.5-flash")
resposta_llm = model.generate_content(prompt)

print("Fontes:", resultados["metadatas"][0])
```

Trade-offs e Limitações Iniciais

- O RAG descrito ("Naïve RAG" ou RAG Ingênuo) resolve 80% dos problemas, mas não é perfeito.
- **A Qualidade Depende do Retrieval:** Se a busca semântica falhar em trazer o parágrafo certo, a resposta do LLM será "Não sei", resultando num sistema excessivamente restritivo.
- **Lost in the Middle:** Se fornecermos $K = 20$ parágrafos recuperados, estudos comprovam empíricamente que os LLMs tendem a esquecer informações críticas posicionadas no "meio" do *prompt*, lembrando apenas do início e do final.

O Pipeline RAG Completo

Cada Etapa Tem um Nome

Retrieval (buscar no banco) → **A**ugmentation (montar o prompt) → **G**eneration (chamar o LLM). Daí o nome: **RAG**.

Alucinação: Anatomia do Problema

O Mecanismo

O LLM foi treinado para **soar plausível**, não para ser factual. O RAG resolve isso **injetando fatos reais** no prompt, forçando o modelo a fundamentar a resposta.

Similaridade de Cosseno: Exemplo Numérico

Como o ChromaDB encontra o chunk mais relevante?

Vetores simplificados (2D):

- Pergunta: $\vec{q} = [0.8,\ 0.6]$ ("multa rescisão")
- Chunk A: $\vec{a} = [0.7,\ 0.7]$ ("multa por rescisão é de 20%")
- Chunk B: $\vec{b} = [0.1,\ 0.9]$ ("férias devem ser avisadas")

$\cos(\vec{q}, \vec{a}) = \frac{0.8 \times 0.7 + 0.6 \times 0.7}{1.0 \times 0.99} = \frac{0.98}{0.99} \approx \mathbf{0.99}$ ✓ Muito similar!

$\cos(\vec{q}, \vec{b}) = \frac{0.8 \times 0.1 + 0.6 \times 0.9}{1.0 \times 0.91} = \frac{0.62}{0.91} \approx \mathbf{0.68}$ × Menos relevante

Resultado

ChromaDB retorna Chunk A ($0.99 > 0.68$). O LLM receberá "multa por rescisão é de 20%" como contexto.

O Efeito do K: Quantos Chunks Enviar?

K	Vantagem	Desvantagem
$K = 1$	Menor custo de tokens	Pode perder contexto crucial
$K = 3$	**Equilíbrio ideal**	Pequeno aumento de custo
$K = 5$	Cobertura ampla	Lost in the Middle
$K = 20$	Quase tudo presente	Estoura contexto + custos

Regra Prática

Comece com $K = 3$. Se a resposta estiver incompleta, suba para $K = 5$. Nunca use $K > 10$ sem **reranking** (reordenação por relevância).

Anatomia do Prompt RAG

Template Rígido

```
SYSTEM: Você é um assistente corporativo.
Responda APENAS com base no contexto abaixo.
Se não souber, diga "Não encontrei nos documentos."

<contexto>
{chunks_recuperados}
</contexto>

USER: {pergunta_do_usuario}
```

3 Elementos Obrigatórios

1) Instrução de âncora: "APENAS com base no contexto".
2) Guardrail: "Se não souber, diga X".
3) Delimitadores: `<contexto>` para separar dados de instrução.

Citação Rastreável: De Onde Veio a Resposta?

Com Metadados

```
resultados["metadatas"][0]
→ {"fonte": "RH",
"pagina": 2}
```

- O usuário sabe **de qual PDF** veio a informação.
- Auditoria e compliance possíveis.

Sem Metadados

O LLM responde "A multa é 20%" sem citar fonte.

Problema: impossível verificar. O jurídico da empresa rejeita.

Boa Prática

Sempre armazene `fonte`, `pagina` e `data` como metadados no ChromaDB.

Recap — O Que Levamos da Aula 17

1. **Alucinação** é inerente ao LLM — ele gera texto plausível, não factual.
2. **RAG** (Retrieve → Augment → Generate) ancora o LLM em dados reais.
3. **ChromaDB** armazena vetores e busca por similaridade de cosseno.
4. **Top-K** controla quantos chunks o LLM recebe (comece com $K = 3$).
5. **Guardrails** obrigam o LLM a dizer "não sei" quando o contexto não cobre.
6. **Metadados** tornam as respostas rastreáveis e auditáveis.

Conexão com a Próxima Aula

Esse RAG "manual" funciona em Python. Mas empresas usam Java/Spring. Na próxima aula, fazemos a **virada de chave** para o ecossistema corporativo.

Roteiro do Laboratório 17

Meta do Lab

Construir um RAG completo do zero em Python: ChromaDB populado com dados de RH, busca semântica, prompt com guardrails, e resposta fundamentada com citação de fonte.

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: Prova Teórica

100% da Aula 17 Concluída!