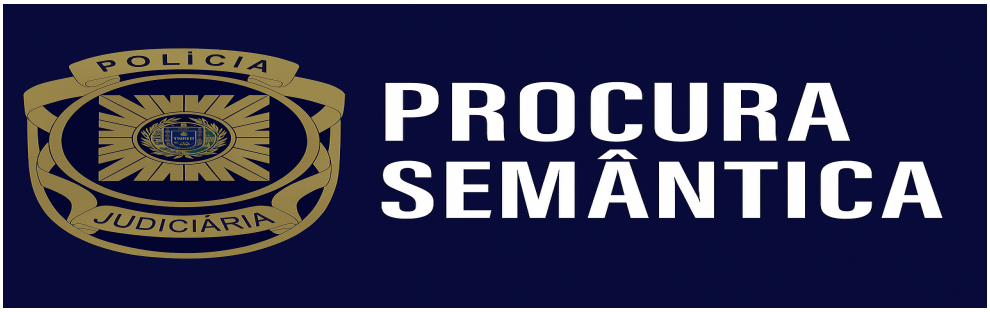
Instituto Superior de Engenharia de Lisboa (ISEL)
Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores (DEETC)

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

Desenvolvimento de uma API de Pesquisa Semântica para Registos Criminais



Pedro Miguel Tavares da Silva Pato e Silva (49734)

Orientador

Professor [Doutor] Helder Filipe de Oliveira Bastos

Junho, 2025

Resumo

Motivado pelo interesse na área criminal e pela necessidade de melhorar a recuperação de informação em registros criminais, este projeto desenvolveu uma API de pesquisa semântica que utiliza processamento de linguagem natural (*NLP*) para interpretar o contexto das consultas. Implementado em *Docker*, o sistema foi programado em *Python* e utiliza uma interface desenvolvida com *Flask* e *React*. Após os testes realizados, o modelo escolhido foi o *medcpt_article*, que alcançou uma precisão de 95,1% em testes realzidados para este projeto. A arquitetura baseada em contentores permite uma gestão dinâmica de dados, com seleção de *buckets* e mecanismos de *Healthcheck*. Desafios como a transição para *Docker* e a optimização do *chunking* foram superados de forma iterativa. O trabalho futuro prevê o *fine-tuning* do modelo em colaboração com a Polícia Judiciária.

Abstract

Driven by an interest in the criminal domain and the need to improve information retrieval in criminal records, this project developed a semantic search API that leverages natural language processing (NLP) to interpret the context of user queries. The system is implemented using *Docker*, programmed in *Python*, and features an interface built with *Flask* and *React*. Following extensive testing, the selected model was *medcpt_article*, which achieved an accuracy of 95.1% in evaluations made for this project. The container-based architecture enables dynamic data management, including the selection of *buckets* and the use of *Healthcheck* mechanisms. Challenges such as the migration to *Docker* and the optimization of *chunking* were addressed iteratively. Future work includes fine-tuning the model in collaboration with the Portuguese Criminal Police (*Polícia Judiciária*).

Agradecimentos

Agradeço ao Professor Doutor Helder Filipe de Oliveira Bastos pelas aulas da unidade curricular de Projeto, que, apesar de diferentes, contribuíram para um excelente aproveitamento das aulas, e pela orientação inestimável e apoio contínuo, fundamentais para o sucesso deste projeto. Agradeço à Polícia Judiciária, cuja área de atuação motivou este trabalho e forneceu importante contexto, assim como aos meus amigos e namorada pela inspiração e discussões técnicas

Ao Professor Doutor Helder Bastos, pela orientação e apoio indispensáveis ao longo deste projeto.

Índice

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A recuperação de informação em registos criminais é um desafio crítico para sistemas judiciais e de segurança pública, devido à complexidade e o volume dos dados. As abordagens tradicionais baseadas em palavras-chave frequentemente falham em captar o contexto semântico das consultas,o que leva a resultados irrelevantes ou incompletos.

Este projeto, motivado por um interesse pessoal na área criminal e pela necessidade de modernizar sistemas de busca, desenvolveu uma API de pesquisa semântica ,focada em relatórios criminais, que utiliza técnicas de processamento de linguagem natural (NLP) para interpretar consultas e recuperar documentos relevantes com base na similaridade semântica.

Inicialmente, considerei escolher o projeto de Retrieval-Augmented Generation (RAG), mas, após uma discussão com o orientador, percebi que não correspondia aos meus objetivos, pois o foco do RAG é gerar respostas baseadas em documentos recuperados, enquanto eu buscava priorizar a recuperação semântica direta.

Sugeri então a pesquisa semântica, que se revelou mais adequada.

A API, implementada com Python, Flask, React, Ollama e Docker, utiliza o modelo medcpt-article, selecionado após testes comparativos com sete modelos, alcançando uma similaridade média de 95,1% em cerca de 100 relatórios criminais em português.

A arquitetura da API é baseada em microserviços, um modelo onde cada componente (como os buckets ou a interface) operam de forma independente, comunicando via interfaces bem definidas, o que facilita escalabilidade e manutenção.

Os utilizadores podem selecionar qualquer bucket para consulta, e a API retorna os k resultados mais relevantes, suporta por mecanismos de Health-check para garantir robustez.

Desafios como a transição para Docker, optimização de chunking e resolução de problemas como resultados duplicados ou incompletos foram superados com apoio do orientador.

Do ponto de vista da engenharia, o projeto seguiu uma abordagem iterativa, com foco na modularidade, escalabilidade e validação rigorosa. Testes extensivos avaliaram métricas como similaridade, tempo de consulta e tempo de processamento de documentos, confirmando a superioridade do medcpt-article.

Este trabalho contribui para a modernização de sistemas judiciais, com potencial aplicação em outros domínios, como relatórios médicos ou jornalísticos.

Este relatório está organizado da seguinte forma:

- **Capítulo 2 - Trabalho Relacionado**: Contextualiza o projeto em relação a sistemas de busca semântica e gestão de registos criminais.
- **Capítulo 3 - Modelo Proposto**: Detalha os requisitos, fundamentos tecnológicos e abordagem metodológica.
- **Capítulo 4 - Implementação do Modelo**: Descreve a arquitetura, componentes e desafios técnicos.
- **Capítulo 5 - Validação e Testes**: Apresenta os resultados dos testes e análises comparativas.
- **Capítulo 6 - Conclusões e Trabalho Futuro**: Resume os resultados e sugere direções futuras.

Capítulo 2

Polícia Judiciária e Pesquisa Semântica

2.1 Introdução à Polícia Judiciária

A Polícia Judiciária (PJ), criada em 1945 pelo Decreto-Lei n.º 35042, é o órgão superior de polícia criminal em Portugal, subordinado ao Ministério da Justiça. Investiga crimes graves, como crime organizado, terrorismo, tráfico de drogas, corrupção e crimes financeiros. A sua estrutura inclui diretorias regionais e unidades especializadas, como a Unidade Nacional de Combate ao Terrorismo e o Laboratório de Polícia Científica.

2.2 Registos Criminais na PJ

Os registos criminais em Portugal, geridos pela Direção-Geral da Administração da Justiça (DGAJ), contêm informações sobre condenações de indivíduos maiores de 16 anos. A PJ acede a bases de dados internas com relatórios detalhados, evidências e perfis de suspeitos, essenciais para investigações. Esses registos, que abrangem crimes complexos como fraude financeira, incluem relatórios textuais, documentos financeiros e evidências forenses.

2.3 Desafios na Pesquisa de Registos

Os sistemas tradicionais de pesquisa baseados em palavras-chave apresentam limitações para a PJ:

- **Dependência de Palavras-Chave**: Exige termos exatos, o que dificulta procuras em documentos extensos.
- **Falta de Contexto**: Não compreende o significado semântico, retornando resultados irrelevantes.
- **Volume de Dados**: Grandes quantidades de dados digitais sobrecarregam as pesquisas tradicionais.
- **Dados Interconectados**: Casos exigem compreender relações entre documentos.

2.4 Compreensão da Pesquisa Semântica

2.4.1 Introdução à Pesquisa Semântica

A pesquisa semântica é uma técnica avançada que visa compreender a intenção e o significado contextual de uma consulta, indo além da correspondência exata de palavras-chave.

Utiliza processamento de linguagem natural (NLP) e AI, interpreta a semântica das palavras, recupera resultados conceptualmente relevantes, mesmo sem termos idênticos.

Por exemplo, uma consulta sobre ?crimes que envolvem computadores? pode corresponder a documentos sobre "cibercrime", porque reconhece a similaridade conceptual.

Essa abordagem é valiosa em domínios com dados complexos, como registos criminais, onde documentos extensos e jargões específicos dificultam procuras tradicionais.

A pesquisa semântica usa *embeddings* ? representações numéricas de texto que capturam significado ? para medir a similaridade entre as consultas e os documentos.

2.4.2 Pesquisa Semântica no Projeto

Neste projeto, a pesquisa semântica é o foco para a API de recuperação de registos criminais. A API utiliza o modelo `medcpt_article` que alcançou 95,1% de precisão nos testes.

Este modelo gera *embeddings* para consultas e documentos, comparados via similaridade de cosseno, que mede a proximidade semântica.

Por exemplo, uma consulta sobre ?crimes cibernéticos? retorna documentos sobre "crimes digitais?, mesmo sem correspondência exata.

A API esta preparada para erros ortográficos, como ?actividades? em vez de ?atividades? porque mantém os resultados relevantes ao focar no conteúdo semântico.

Capítulo 3

Trabalho Relacionado

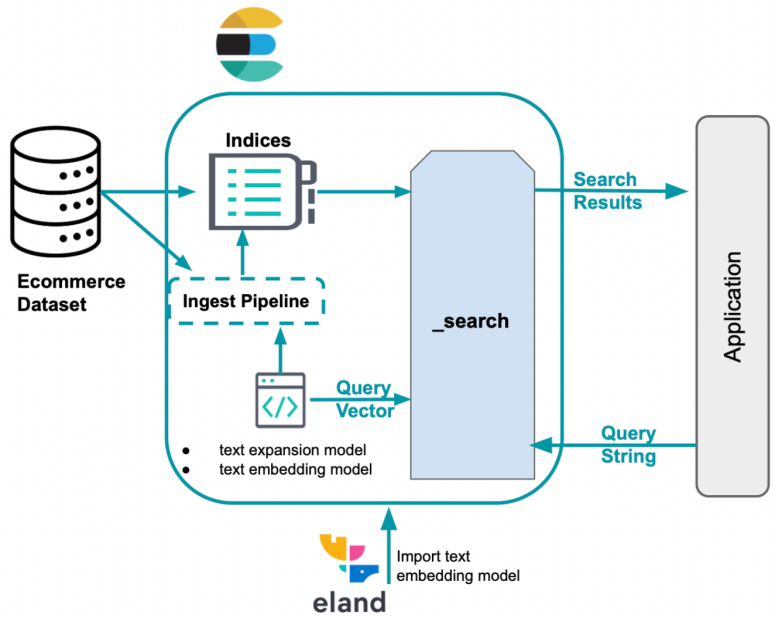


Figura 3.1: Arquitetura do Elasticsearch com extensões de NLP.

A pesquisa semântica tem sido explorada em sistemas de procura de informação, mas a sua aplicação em registos criminais é limitada. O Elasticsearch, com extensões de NLP, suporta buscas avançadas, mas não é optimizado para textos jurídicos complexos.

Figura 3.2: Crachá da Polícia Judiciária.

Sistemas tradicionais, como os da Polícia Judiciária, dependem de buscas por palavras-chave, limitando a relevância dos resultados.

A API desenvolvida neste projeto distingue-se pelo modelo medcpt-article, optimizado para textos longos, e pela arquitetura Docker, que permite escalabilidade e gestão dinâmica de dados. Apesar de treinado em relatórios médicos, o medcpt-article demonstrou excelente desempenho em relatórios criminais devido à sua capacidade de captar semântica em textos longos e estruturados. A capacidade de selecionar buckets e os mecanismos de Healthcheck tornam a solução robusta, contribuindo para avanços na procura de informação jurídica.

Capítulo 4

Modelo Proposto

Este capítulo apresenta o modelo proposto para a API de pesquisa semântica, delineando os requisitos, fundamentos tecnológicos e a abordagem metodológica, com o objetivo de fornecer uma visão clara dos objetivos do projeto.

4.1 Requisitos

Os requisitos do sistema foram definidos para garantir funcionalidade e robustez, conforme apresentado nas Tabelas 4.1 e 4.2.

Nº	**Requisito**	**Implementado**
1	Pesquisa semântica em linguagem natural	Sim
2	Interface de utilizador para consultas	Sim
3	Seleção de *buckets* para consultas	Sim
4	*Healthcheck* aos *buckets*	Sim
5	Feedback sobre resultados	Sim

Tabela 4.1: Requisitos Funcionais

Nº	Requisito	Implementado
1	Escalabilidade para múltiplos *buckets*	Sim
2	Monitorização de logs	Sim
3	Segurança de dados	Não

Tabela 4.2: Requisitos Não Funcionais

4.2 Fundamentos

A API utiliza técnicas de NLP para processar consultas em linguagem natural, que geram embeddings com o modelo `medcpt_article` via Ollama. Python e Flask formam o backend, enquanto React suporta a interface de utilizador. Docker permite a execução de buckets independentes, e FAISS é usado para indexação e busca de embeddings. A escolha do `medcpt_article` foi baseada em sua capacidade de lidar com textos longos, como relatórios criminais como vamos observar nos resultados do teste.

4.3 Abordagem

O desenvolvimento seguiu uma metodologia iterativa:

- **Prototipagem Local**: Inicialmente, o projeto foi desenvolvido localmente para explorar as técnicas de embeddings e de pesquisa semântica.
- **optimização de Chunking**: Passou-se de chunking baseado em caracteres para palavras, melhorando a relevância dos resultados.
- **Transição para Docker**: Adotou-se uma arquitetura baseada em contentores para escalabilidade e gestão dinâmica.
- **Seleção de Modelo**: Sete modelos foram testados, com o `medcpt_article` selecionado por sua precisão de 95,1%.

4.4 Modelo Utilizado

O modelo utilizado neste sistema é o medcpt-article, projetado para processar textos longos de forma eficiente e treinado em relatórios médicos reais. Diferente do modelo nomic-embed-text, que foi inicialmente considerado e optimizado para consultas curtas, o "medcpt-article"oferece melhor desempenho em documentos extensos, capturando contextos mais amplos nos embeddings gerados.

Capítulo 5

Implementação do Modelo

5.1 Arquitetura

A API adota uma arquitetura de microserviços, com Flask para endpoints REST, React para a interface, e Ollama para executar o medcpt-article em contentores Docker. Cada bucket é um serviço independente, configurado via `docker-compose.yml`, que permite adicionar dados sem interrupção.

- **Interface**: Desenvolvida em React, permite a seleção de buckets e a submissão de consultas em linguagem natural.

- **Buckets**: Serviços independentes que geram e armazenam embeddings dos documentos, processando consultas em paralelo.

- **AI Node**: Coordena o sistema, registra buckets, faz /textitHealth-checks e encaminhando consultas para processamento.

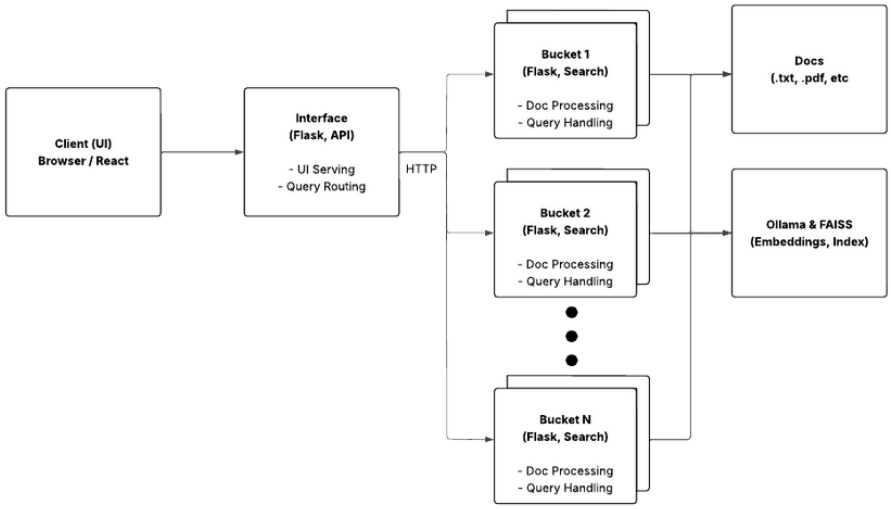


Figura 5.1: Arquitetura da API de Pesquisa Semântica

5.1.1 Arquitetura da Interface

A interface utiliza componentes React para interação dinâmica, incluindo um seletor de buckets e uma área de entrada de texto. O *AI Node* é integrado como um ponto de comunicação central, recebe consultas e retornando respostas processadas. O *AI Node* é uma parte integral da interface, serve como o centro de comunicação entre a interface e os *buckets* como mencionado anteriormente.

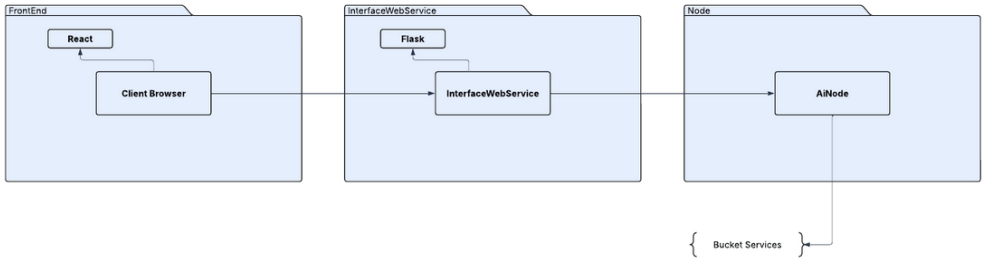


Figura 5.2: Arquitetura da interface.

5.1.2 Arquitetura dos Buckets

Os buckets são serviços modulares que dividem documentos em chunks de 300 palavras, geram embeddings com o modelo "medcpt-article"e os armazenam-nos para consultas futuras. Cada bucket opera de forma independente, permitindo escalabilidade.

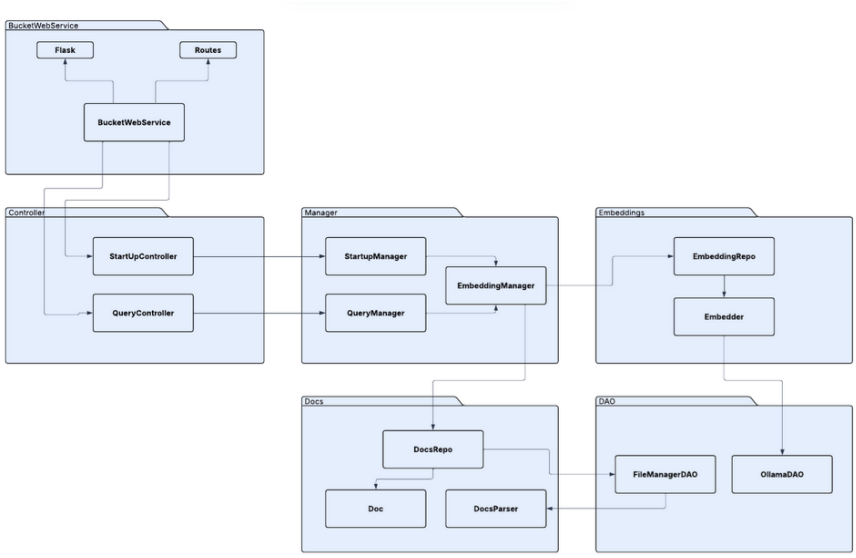


Figura 5.3: Arquitetura dos buckets.

5.1.3 Componentes Principais

Os componentes principais do sistema foram projetados para fornecer capacidades de pesquisa semântica, que utilizam *embeddings*, uma arquitetura distribuída de *buckets* e uma interface para interação com o utilizador. Abaixo, detalhamos cada componente e suas funcionalidades.

Criação e Armazenamento de *Embeddings*

Os *embeddings* são representações numéricas de texto que capturam o significado semântico, que permite pesquisas baseadas em similaridade. Este processo envolve duas classes principais: `Embedder.py` e `EmbeddingRepo.py`.

`Embedder.py` A classe `Embedder` gera *embeddings* a partir de texto utilizando o serviço Ollama.

- **Divisão de Texto**: Documentos grandes são divididos em *chunks* para garantir que os *embeddings* representem o contexto local de forma eficaz. O método `chunk_content` divide o texto com base no número de palavras, com tamanho e sobreposição configuráveis.

```
def chunk_content(self, content: str,
    chunk_size_words: int = 300, overlap_words: int =
     150) -> List[str]:
     words = content.split()
     chunks = []
     for i in range(0, len(words) - chunk_size_words
        + 1, chunk_size_words - overlap_words):
          chunk_words = words[i:i + chunk_size_words]
          chunk_text = ' '.join(chunk_words)
          chunks.append(chunk_text)
     if len(words) > chunk_size_words and (len(words)
          - i - 1) > overlap_words:
          chunk_words = words[-chunk_size_words:]
          chunks.append(' '.join(chunk_words))
     return chunks
```

Código 1: Método de chunking em `Embedder.py`

- **Geração de *Embeddings***: O método `generate_embedding` envia um *chunk* de texto ao serviço Ollama via `OllamaDAO` e retorna o *embedding* como um array NumPy.

```
def generate_embedding(self, text: str) -> np.
    ndarray:
     return self.ollama_dao.generate_embedding(text)
```

Código 2: Geração de embedding em `Embedder.py`

`EmbeddingRepo.py` A classe `EmbeddingRepo` lida com o armazenamento e recuperação de *embeddings* utilizando o FAISS, uma biblioteca otimizada para pesquisa de similaridade.

- **Configuração do Índice**: Utiliza um índice `IndexFlatIP` da FAISS com dimensão 768 (correspondente ao tamanho do *embedding*) para pesquisa de similaridade por produto interno.

```
self.faiss_index = faiss.IndexFlatIP(768)
```

Código 3: Configuração do índice FAISS

- **Armazenamento de *Embeddings***: O método `save` normaliza os *embeddings* (para garantir magnitude consistente) e os adiciona ao índice FAISS, associando-os aos metadados do documento.

```
def save(self, doc: Doc, chunks: List[str] = None):
    for i, embedding in enumerate(doc.embeddings):
        norm = np.linalg.norm(embedding)
        normalized_embedding = embedding / norm if
           norm != 0 else embedding
        self.faiss_index.add(np.array([
           normalized_embedding]))
        chunk_text = chunks[i] if chunks and i < len
           (chunks) else ""
        self.docs.append((doc, i, chunk_text))
```

Código 4: Armazenamento de embeddings

- **Pesquisa de *Embeddings***: O método `search` encontra os k *embeddings* mais similares a um vetor de consulta, retorna os detalhes do documento e pontuações de similaridade.

```
def search(self, query_vector: np.ndarray, k: int =
   3) -> List[Tuple[Doc, float, int, str]]:
    norm = np.linalg.norm(query_vector)
    normalized_query = query_vector / norm if norm
       != 0 else query_vector
    search_k = min(k * 2, self.faiss_index.ntotal)
    distances, indices = self.faiss_index.search(np.
       array([normalized_query]), search_k)
    similarities = distances[0]
    results = []
    seen_docs = set()
    for j, i in enumerate(indices[0]):
        if len(results) >= k:
            break
        doc, chunk_index, chunk_text = self.docs[i]
        if doc.name not in seen_docs:
            results.append((doc, similarities[j],
               chunk_index, chunk_text))
            seen_docs.add(doc.name)
    return results
```

Código 5: Pesquisa de embeddings

Gestão de *Buckets*

Os *buckets* são unidades independentes que processam e armazenam *embeddings* de documentos. A sua gestão envolve registo, processamento de documentos e manipulação de ficheiros.

`BucketWebService.py` A classe `BucketWebService` gere o serviço web do *bucket* e sua interação com o *AI Node*.

- **Configuração**: Utiliza variáveis de ambiente para definir o nome, pasta e URLs de comunicação do *bucket*.

```
self.bucket_name = os.getenv("BUCKET_NAME", "
    default_bucket")
self.bucket_folder = os.getenv("BUCKET_FOLDER", "./
    documents")
self.ai_node_url = os.getenv("AI_NODE_URL", "http://
    interface:5000/ai-node")
self.bucket_url = os.getenv("BUCKET_URL", "http://
    bucket1:5000")
```

Código 6: Configuração do bucket

- **Registo**: Os *buckets* registam-se no *AI Node* enviando seu nome e URL, acionando o processamento de documentos após o registo bem-sucedido.

```
def _register_with_ai_node(self, app):
    response = requests.post(
        f"{self.ai_node_url}/register",
        json={"name": self.bucket_name, "url": self.
            bucket_url}
    )
    response.raise_for_status()
    with app.app_context():
        result = self.startup_controller.startup()
    self.set_processing_complete(True)
    app.processing_complete = True
```

Código 7: Registo do bucket

`StartUpController.py` A classe `StartUpController` processa os documentos no *bucket* durante a inicialização.

- **Processamento de Documentos**: Corre a pasta do *bucket* em busca de tipos de ficheiros suportados (e.g., `.txt`, `.pdf`) e processa-os utilizando o `EmbeddingManager`.

```
def startup(self):
    bucket_folder = os.getenv('BUCKET_FOLDER', './
       documents')
    for root, _, files in os.walk(bucket_folder):
        for filename in files:
            if os.path.splitext(filename)[1].lower()
                in supported_extensions:
                file_path = os.path.join(root,
                   filename)
                self.embedding_manager.
                   process_document(file_path)
    return jsonify({'message': f'Processados␣{
       document_count}␣documentos', 'startup_time':
       total_time})
```

Código 8: Processamento de documentos

`FileManagerDAO.py` A classe `FileManagerDAO` lida com o acesso a ficheiros dentro dos *buckets*.

- **Listagem de Ficheiros**: Recupera uma lista de ficheiros suportados do diretório do *bucket* e suas subpastas.

```
def get_docs(self) -> List[Dict[str, str]]:
    docs = []
    for root, _, files in os.walk(self.directory):
        for filename in files:
            if any(filename.lower().endswith(ext)
               for ext in self.SUPPORTED_EXTENSIONS)
               :
                file_path = os.path.join(root,
                   filename)
                docs.append({
                    "name": filename,
                    "location": os.path.relpath(
                       file_path, self.directory),
```

```
                    "content": ""
                })
    return docs
```

Código 9: Listagem de ficheiros

Gestão de Consultas

As consultas são processadas para encontrar documentos semanticamente similares com o uso dos *embeddings*.

`QueryController.py` A classe `QueryController` lida com requisições de consulta via endpoint web.

- **Endpoint de Consulta**: Aceita requisições POST com uma string de consulta e um valor k, retornando resultados classificados.

```
@app.route("/query", methods=["POST"])
def query_endpoint():
    data = request.get_json()
    query_text = data.get("query", "")
    k = int(data.get("k", 3))
    result = self.embedding_manager.process_query(
       query_text, k)
    return jsonify({
        "results": result["results"],
        "query_time": query_time
    })
```

Código 10: Endpoint de consulta

`QueryManager.py` A classe `QueryManager` processa consultas que geram *embeddings* e faz uma pesquisa no repositório.

- **Processamento de Consultas**: Converte a consulta num *embedding* e recupera as k melhores correspondências.

```
def process_query(self, query: str, k: int = 3) ->
   List[Dict[str, any]]:
    query_vector = self.embedding_manager.embedder.
       generate_embedding(query)
```

```
    results = self.embedding_repo.search(
       query_vector, k)
    result_list = [
        {
            "name": doc.name,
            "similarity": float(dist),
            "location": doc.location,
            "chunk": chunk_text,
        }
        for doc, dist, chunk_index, chunk_text in
           results
    ]
    return result_list
```

Código 11: Processamento de consultas

Healthcheck e Comunicação

O sistema garante confiabilidade através de *healthchecks* e comunicação eficiente entre *buckets* e a interface.

`AINode.py` A classe `AINode` supervisiona a saúde dos *buckets* e encaminha consultas.

- ***Healthcheck***: Verifica o estado dos *buckets* a cada 30 segundos, envia uma requisição GET para o endpoint `/status`. Um *bucket* é marcado como vivo se responder com código 200.

```
def healthcheck(buckets):
    while True:
        for bucket in buckets:
            try:
                r = requests.get(f"{bucket['url']}/
                   status", timeout=2)
                bucket['alive'] = r.status_code ==
                   200
                bucket['processing_complete'] = r.
                   json().get('processing_complete',
                    False)
            except Exception:
                bucket['alive'] = False
                bucket['processing_complete'] =
                   False
```

```
        time.sleep(30)
```

Código 12: Healthcheck dos buckets

- **Encaminhamento de Consultas**: Encaminha consultas de forma assíncrona para os *buckets* selecionados e agrega os resultados.

```
async def forward_query(self, query, k,
   selected_buckets):
    results = {}
    async with httpx.AsyncClient() as client:
        for bucket in self.buckets:
            if bucket['name'] in selected_buckets:
                try:
                    url = f"{bucket['url']}/query"
                    resp = await client.post(url,
                       json={"query": query, "k": k
                       }, timeout=10)
                    if resp.status_code == 200:
                        data = resp.json()
                        results[bucket['name']] =
                           data.get("results", [])
                    else:
                        results[bucket['name']] = [{
                           "error": f"Bucket␣
                           retornou␣estado␣{resp.
                           status_code}"}]
                except Exception as e:
                    results[bucket['name']] = [{"
                       error": f"Bucket␣inacessível:
                       ␣{str(e)}"}]
    return results
```

Código 13: Encaminhamento de consultas

`OllamaDAO`

A classe `OllamaDAO` integra-se com o serviço Ollama para gerar *embeddings*.

- **Inicialização**: Conecta-se ao serviço Ollama com um host e modelo configuráveis.

```
def __init__(self):
```

```
    self.model = "nomic-embed-text:v1.5"
    ollama_host = os.getenv("OLLAMA_HOST", "http://
       localhost:11434")
    os.environ["OLLAMA_HOST"] = ollama_host
    try:
        ollama.list()
        logger.info(f"Serviço␣Ollama␣inicializado␣
           com␣o␣modelo␣{self.model}")
    except Exception as e:
        logger.error("Serviço␣Ollama␣indisponível")
        raise RuntimeError("Serviço␣Ollama␣
           indisponível") from e
```

Código 14: Inicialização do OllamaDAO

- **Geração de *Embeddings***: Envia texto ao Ollama e recupera o *embedding* como um array NumPy.

```
def generate_embedding(self, text: str) -> np.
   ndarray:
    try:
        response = ollama.embeddings(model=self.
           model, prompt=text)
        embedding = np.array(response["embedding"],
           dtype=np.float32)
        return embedding
    except Exception as e:
        logger.error(f"Falha␣ao␣gerar␣embedding:␣{
           str(e)}")
        raise
```

Código 15: Geração de embedding via Ollama

Componentes Adicionais

- `DocsParser`: Extrai texto de ficheiros (e.g., `.pdf`, `.docx`) com a bibliotecas PyPDF2 e prepara o conteúdo para *embeddings*.

- `DocsRepo`: Integra `FileManagerDAO` e `DocsParser` para procurar e analisar documentos para criar objetos `Doc` com metadados e conteúdo.

- `EmbeddingManager`: Orquestra o processamento de documentos, coordenando `Embedder` e `EmbeddingRepo` para gerar e armazenar *embeddings*.

- `Interface` (`InterfaceWebService` e `main_interface.py`): Serve um frontend baseado em React e encaminha requisições para *buckets* via *AI Node*, proporcionando uma interface amigável ao utilizador.

- ***Logging***: Registo abrangente em todos os componentes auxilia no monitoramento, depuração e optimização de desempenho.

Este sistema processa documentos, armazena *embeddings* e lida com consultas de forma eficiente, tornando-o adequado para aplicações de pesquisa semântica.

5.2 Métodos da Pesquisa Semântica

5.2.1 Cálculo de Similaridade

A similaridade entre a consulta e os embeddings dos documentos é calculada com a similaridade de cosseno, que mede o cosseno do ângulo entre dois vetores. A fórmula é:

$$\text{similaridade} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

onde $\mathbf{A}$ é o vetor de embedding da consulta e $\mathbf{B}$ é o vetor de embedding do documento.

Por exemplo, se uma consulta como "crimes cibernéticos"tiver o embedding $\mathbf{A} = [1, 0]$ e um documento com "segurança online"tiver $\mathbf{B} = [0.5, 0.5]$, o cálculo é:

$$\text{similaridade} = \frac{1 \cdot 0.5 + 0 \cdot 0.5}{\sqrt{1^2 + 0^2} \cdot \sqrt{0.5^2 + 0.5^2}} = \frac{0.5}{1 \cdot \sqrt{0.5}} \approx 0.707$$

Este valor indica uma similaridade moderada. A comparação final é feita considerando o chunk com a maior pontuação de similaridade em cada documento e não uma média de todos os chunks.

5.2.2 Escolha do Tamanho de Chunk

O tamanho de chunk de 300 palavras foi escolhido após experimentos que equilibraram contexto e eficiência. Chunks menores (e.g., 200 tokens) aumentam a granularidade, mas elevam o custo computacional. Chunks maiores (e.g., 500 tokens) reduzem custos, mas perdem especificidade nos embeddings. O valor de 300 palavras, embora não seja exatamente a média dos tamanhos testados, mostrou-se ideal para otimizar pontuações de similaridade e desempenho do sistema.

5.3 Interface

Durante o desenvolvimento, foram enfrentadas diversas dificuldades técnicas, entre as quais:

1. **Transição para Docker**: A migração da implementação local para contentores exigiu ajustes nas configurações de redes e volumes, resolvidos com o apoio do orientador.

2. **optimização de *Chunking***: A abordagem inicial, baseada no número de caracteres, foi substituída por uma segmentação por palavras, o que melhorou a relevância dos resultados.

3. **Filtro de Duplicados**: A filtragem de documentos repetidos limitava o número de resultados. A solução passou por aumentar o número de candidatos retornados pela FAISS antes de aplicar o filtro.

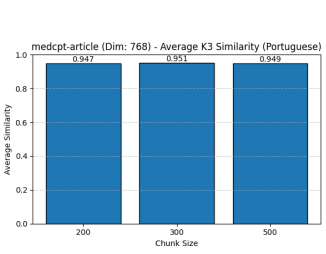
Capítulo 6

Validação e Testes

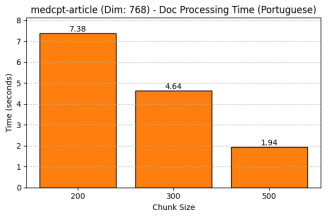
6.1 Gráficos de Desempenho por Modelo

Esta seção apresenta os resultados dos modelos testados (`medcpt-article`, `medcpt-query`, `nomic-embed`, `all-minilm`, `mxbai-embed`, `avrsfr-embed`, `snowflake-embed2`) em documentos em português, com base nas métricas de similaridade média (K3), tempo de processamento de documentos e tempo de consulta K3, para diferentes tamanhos de *chunk* (200, 300, 500 palavras). As figuras de cada modelo são apresentadas lado a lado, seguidas de suas análises.

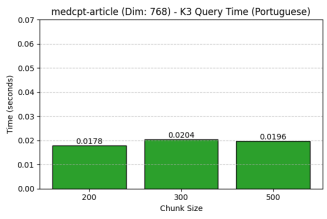
6.1.1 medcpt-article



(a) Similaridade K3



(b) Tempo de processamento



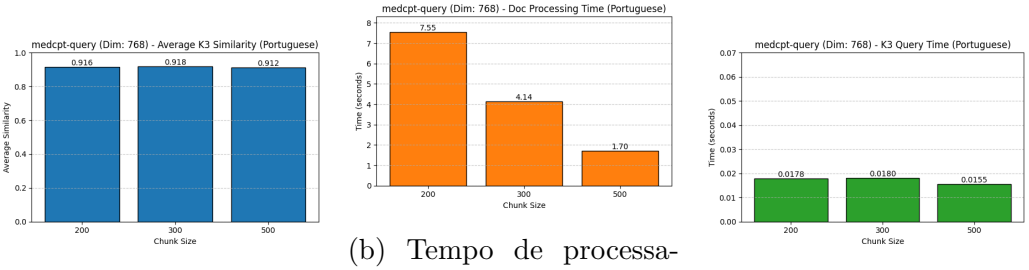
(c) Tempo de consulta K3

Figura 6.1: Desempenho do modelo `medcpt-article`.

O modelo `medcpt-article` apresentou a maior similaridade K3 (0.947, 0.951, 0.949 para *chunk sizes* 200, 300, 500), indicando recuperação de documentos altamente relevante. O tempo de processamento de documentos

diminui com o aumento do *chunk size* (7.38s, 4.64s, 1.94s), devido à redução de *chunks* (4.7, 3.03, 1.22). O tempo de consulta K3 é estável (0.018?0.020s). Comparado ao `medcpt-query`, tem ligeira vantagem em similaridade, com tempos de processamento semelhantes.

6.1.2 medcpt-query

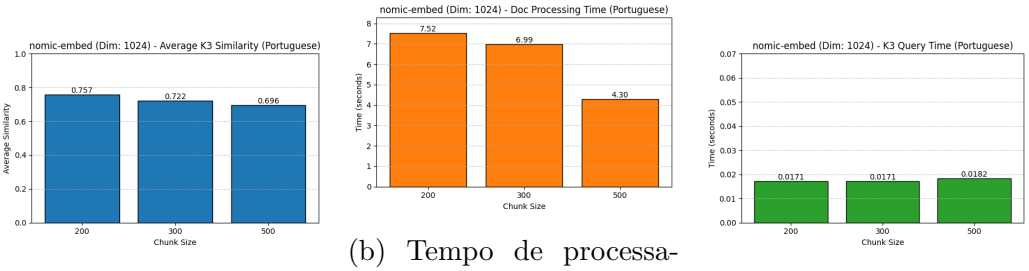


(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.2: Desempenho do modelo `medcpt-query`.

O `medcpt-query` exibiu alta similaridade K3 (0.916, 0.918, 0.912 para *chunk sizes* 200, 300, 500), ligeiramente inferior ao `medcpt-article`. O tempo de processamento de documentos é eficiente (7.55s, 4.14s, 1.70s), similar ao `medcpt-article`, e o tempo de consulta K3 é rápido (0.016?0.018s). Supera o `nomic-embed` em similaridade e precisão, mantendo tempos de consulta competitivos.

6.1.3 nomic-embed

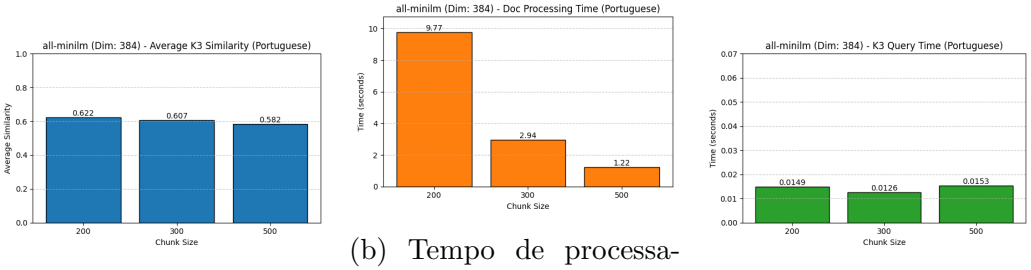


(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.3: Desempenho do modelo `nomic-embed`.

O `nomic-embed` alcançou similaridade K3 moderada (0.757, 0.722, 0.696), inferior aos `medcpt`. O tempo de processamento de documentos é razoável (7.52s, 6.99s, 4.30s), mas ligeiramente superior ao `medcpt-query` para *chunk size* 500. O tempo de consulta K3 é o mais rápido (0.017?0.018s). Comparado ao `mxbai-embed`, tem melhor similaridade e tempos de consulta mais curtos.

6.1.4 all-minilm

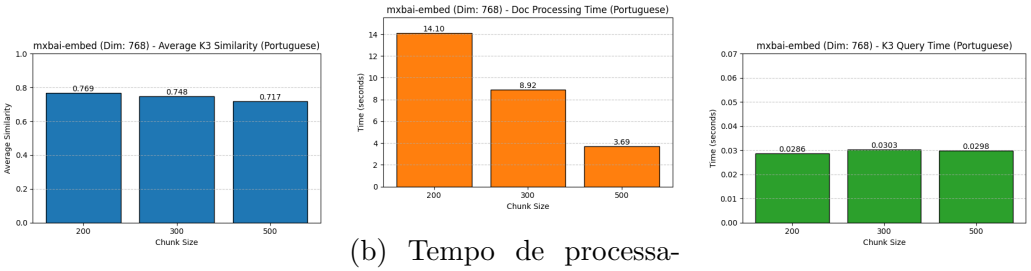


(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.4: Desempenho do modelo `all-minilm`.

O `all-minilm` apresentou a menor similaridade K3 (0.622, 0.607, 0.582), indicando recuperação menos eficaz. O tempo de processamento é competitivo para *chunk size* 500 (1.22s), mas elevado para 200 (9.77s). O tempo de consulta K3 é o mais baixo (0.013?0.015s), mas a baixa similaridade limita sua utilidade. Comparado ao `snowflake-embed2`, tem melhor similaridade.

6.1.5 mxbai-embed

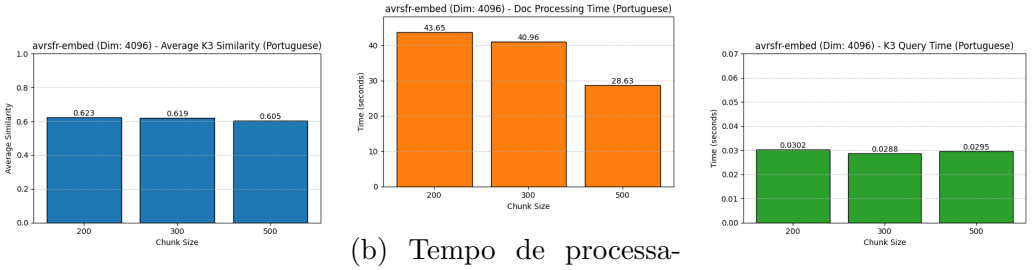


(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.5: Desempenho do modelo `mxbai-embed`.

O `mxbai-embed` obteve similaridade K3 razoável (0.769, 0.748, 0.717), inferior aos `medcpt`. O tempo de processamento é elevado para *chunk size* 200 (14.10s), mas reduz para 500 (3.69s). O tempo de consulta K3 (0.029?0.030s) é mais lento que `nomic-embed` e `medcpt`. Comparado ao `avrsfr-embed`, tem melhor similaridade, mas tempos de processamento mais longos para *chunk size* 200.

6.1.6 avrsfr-embed

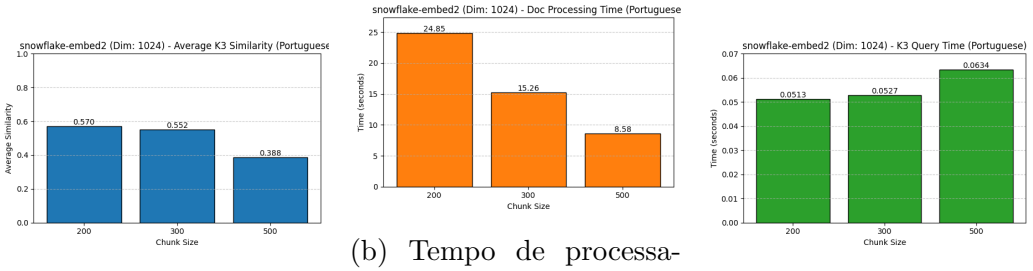


(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.6: Desempenho do modelo `avrsfr-embed`.

O `avrsfr-embed` apresentou similaridade K3 baixa (0.623, 0.619, 0.605). O tempo de processamento é o mais alto (43.65s, 40.96s, 28.63s), devido ao tamanho do embedding (4096). O tempo de consulta K3 (0.029?0.030s) é semelhante ao `mxbai-embed`. Comparado ao `snowflake-embed2`, tem melhor similaridade, mas tempos de processamento superiores.

6.1.7 snowflake-embed2



(a) Similaridade K3 (b) Tempo de processamento (c) Tempo de consulta K3

Figura 6.7: Desempenho do modelo `snowflake-embed2`.

O `snowflake-embed2` teve o pior desempenho em similaridade K3 (0.570, 0.552, 0.388), inadequado para recuperação de documentos. O tempo de processamento é elevado (24.85s, 15.26s, 8.58s), e o tempo de consulta K3 é o mais lento (0.051?0.063s). É consistentemente inferior em todas as métricas.

6.2 Comparação de Modelos

Esta subseção apresenta uma análise comparativa dos modelos, utilizando gráficos gerados pelo script `compare_metrics.py`, destacando diferenças para *chunk size* 300.

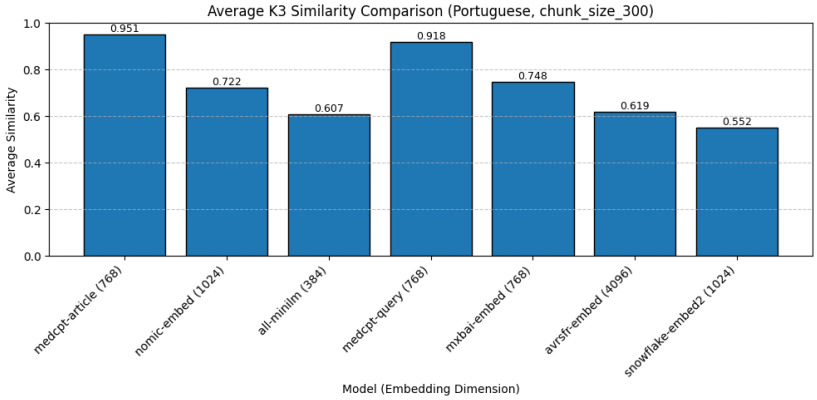


Figura 6.8: Comparação da similaridade média (K3) entre os modelos para *chunk size* 300.

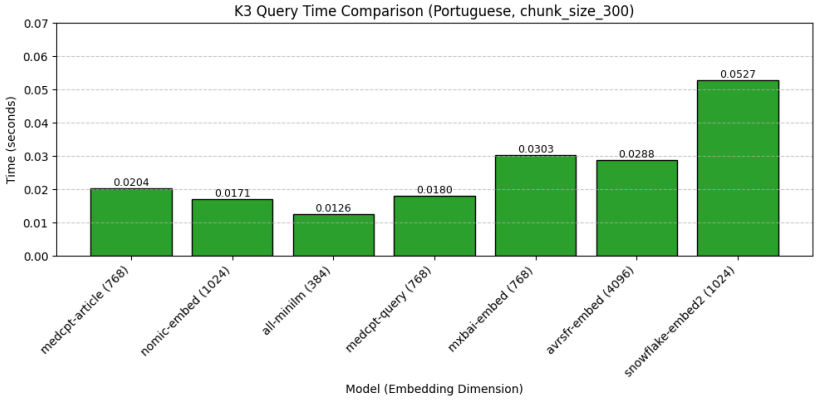


Figura 6.9: Comparação do tempo de consulta K3 entre os modelos para *chunk size* 300.

6.2.1 Comparação Geral

Os modelos `medcpt-article` e `medcpt-query` destacam-se com similaridades K3 (¿0.91), ideais para alta relevância. O `nomic-embed` equilibra si-

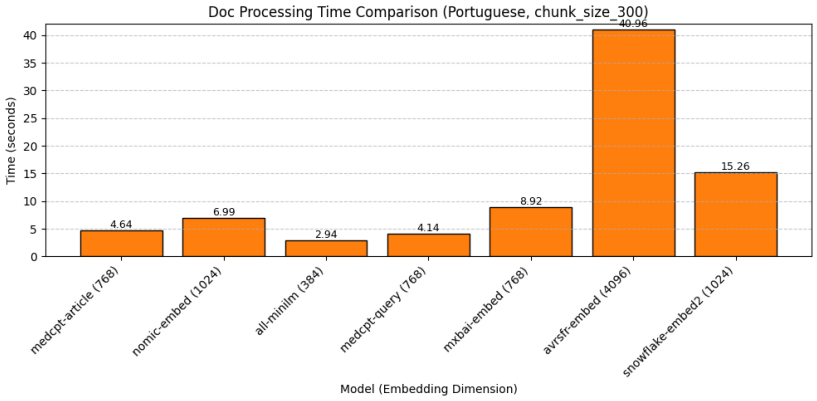


Figura 6.10: Comparação do tempo de processamento de documentos entre os modelos para *chunk size* 300.

milaridade moderada e tempos de consulta rápidos, adequado para velocidade. `all-minilm`, `mxbai-embed`, `avrsfr-embed` e `snowflake-embed2` têm limitações em similaridade ou eficiência. Para *chunk size* 500, o tempo de processamento reduz, mas a similaridade cai, exceto para `medcpt-article` e `medcpt-query`.

6.2.2 Seleção do Modelo `medcpt-article`

O `medcpt-article` foi selecionado devido à alta similaridade K3 (0.951), eficiência em processamento (4.64s) e tempos de consulta ágeis (0.0204s). Embora o `medcpt-query` seja competitivo, seu treino em textos curtos o torna menos adequado para relatórios criminais.

6.3 Consultas e Script de Teste

6.3.1 Consultas Utilizadas

Consultas em português, abrangendo tópicos variados:

- "Quais são os relatórios recentes sobre atividades de gangues armadas em Vilkor, Zakovia?"
- "Quais são os relatórios mais recentes sobre roubos de arte envolvendo os Ghost Shadows em Ravenska, Zakovia?"

- "Quais incidentes envolvendo os esquemas de proteção dos Blood Ravens ocorreram perto de Baron's Peak em 2023?"
- "Quais são os casos de contrabando de armas relatados em Sokovia em 2024?"
- "Quais atividades criminosas dos Night Vipers foram documentadas em Krov, Zakovia?"
- "Quais são as políticas de turismo sustentável em Ravenska, Zakovia?"

6.3.2 Verificação de Robustez a Erros Ortográficos

O script `test_model.py` testou erros ortográficos (e.g., ?actividddades? por ?atividades?). As pontuações de similaridade permaneceram altas, com queda mínima (0.85 para 0.80), mostrando tolerância a erros.

- "Quais são os relatórrios resentes sobbre atividddades de ganguues armadaz em Vilkorr, Zakovia?"
- "Quais são os relatórios mais recntes sobre rouboos de arte enolvendo os Ghoost Shadws em Ravennska, Zakovia?"
- "Quais incidntes envovendo os esquemmas de proteçção dos Blod Ravns ocoreram perto de Baroon?s Peek em 2023?"
- "Quais são os casso de contrabbando de armaz relatadoz em Sokovia em 2024?"
- "Quais atvidades criminossas dos Nigt Viipers foram documntadas em Kroov, Zakovia?"
- "Quais são as políicas de turismmo sustntável em Ravennska, Zakovia?"

6.4 Script `test_model.py`

O script automatiza a avaliação dos modelos:

1. **Processamento de Documentos**: Lê documentos, divide em *chunks* (200, 300, 500 tokens), gera *embeddings* via Ollama.

2. **Indexação com FAISS**: Armazena *embeddings* para busca eficiente.
3. **Execução de Consultas**: Mede similaridade (K3), tempos de consulta e processamento.
4. **Avaliação de Desempenho**: Calcula médias das métricas.
5. **Geração de Resultados**: Exporta para JSON, permitindo análises e gráficos.

6.5 Exemplos de Uso da Interface

A interface em React permite selecionar *buckets*, realizar consultas e visualizar resultados.

Figura 6.11: Tela da interface.

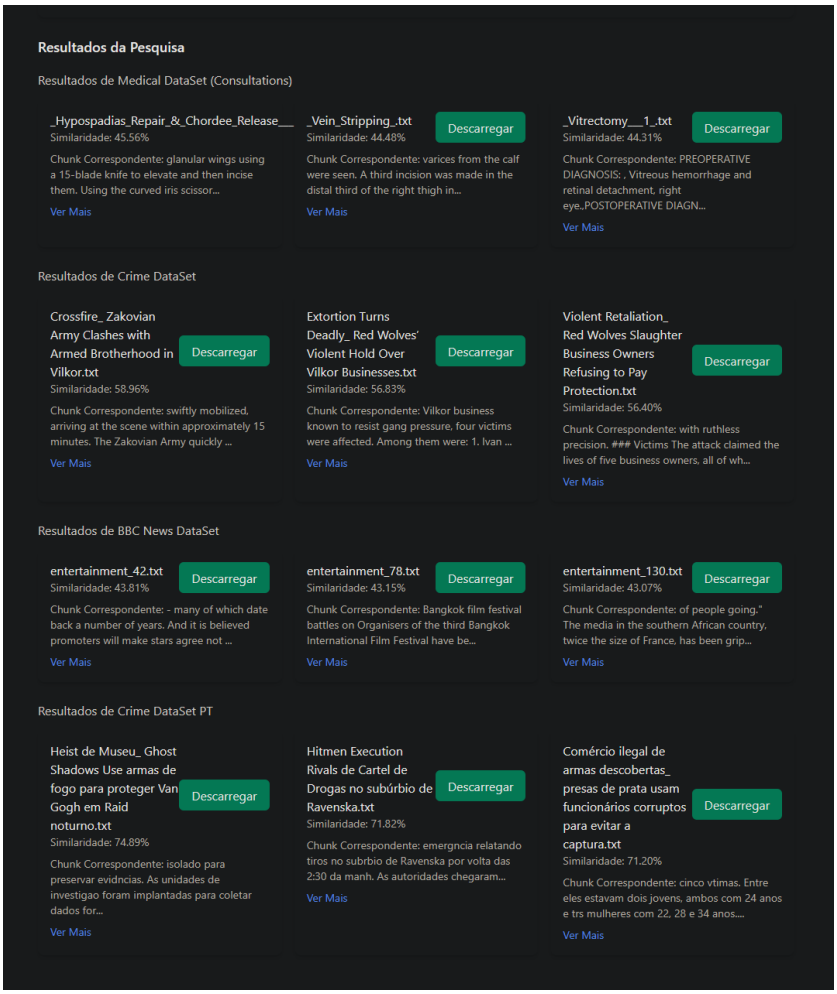


Figura 6.12: Resultado de uma consulta.

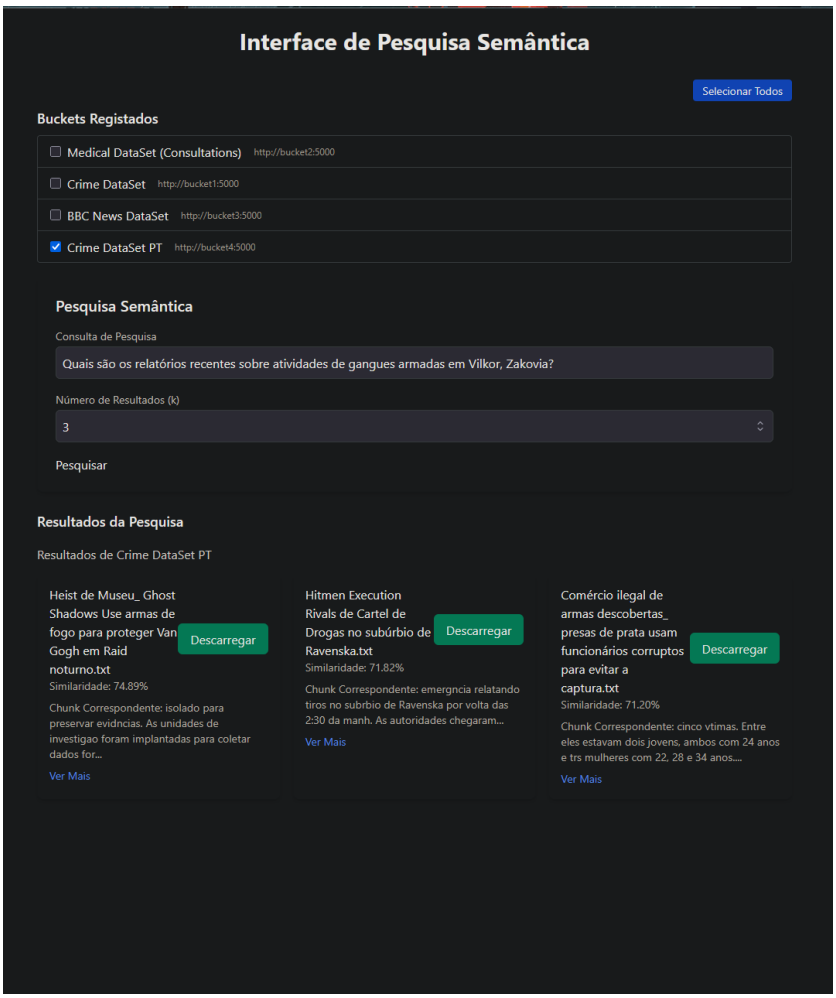


Figura 6.13: Consulta num só *bucket* com $k = 3$.

Figura 6.14: Consulta num só *bucket* com $k = 12$.

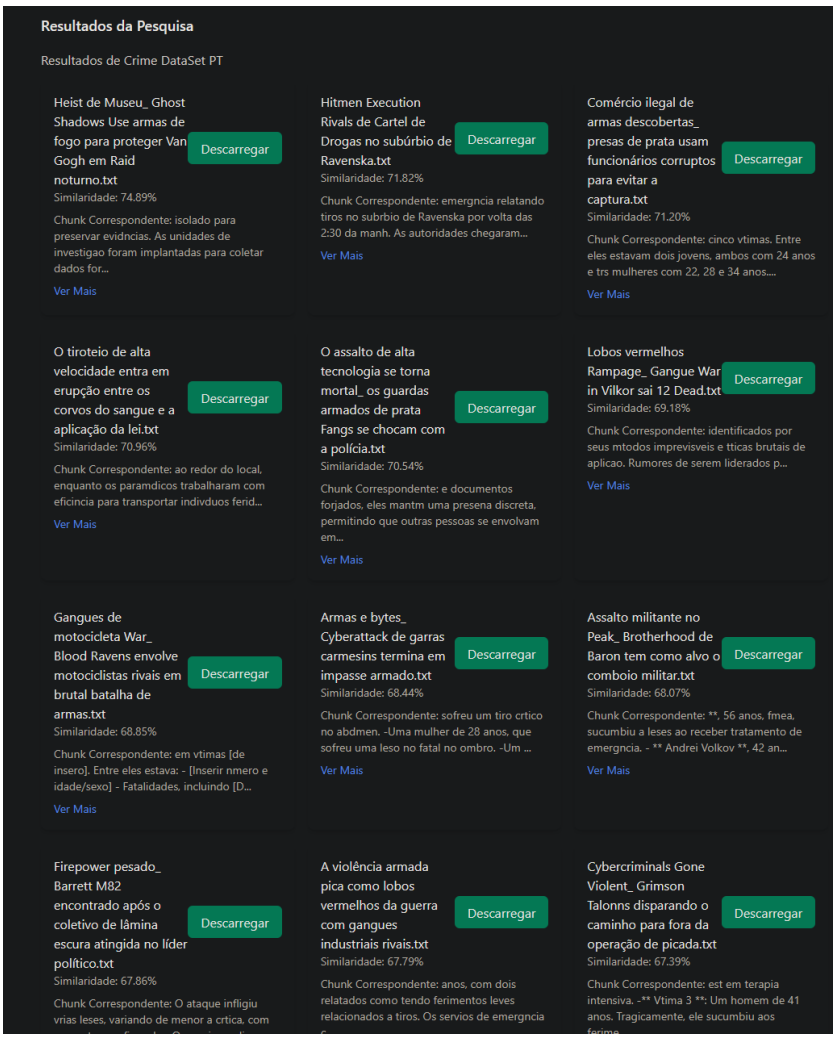


Figura 6.15: Resultados da consulta num só *bucket* com $k = 12$.

Figura 6.16: Demonstração de um *bucket* que falhou.

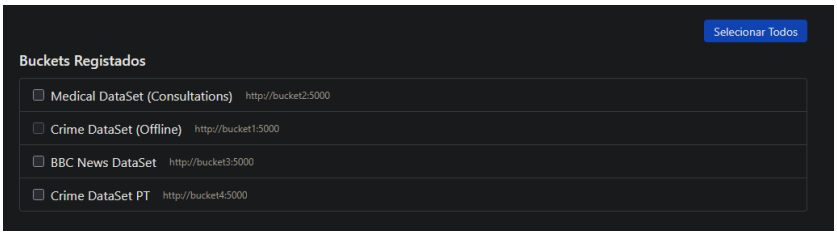


Figura 6.17: Seleção do *bucket* que falhou agora indisponível.

Capítulo 7

Conclusões e Trabalho Futuro

7.1 Conclusão

Este projeto demonstrou, de forma prática e eficaz, como a aplicação de técnicas modernas de processamento de linguagem natural pode transformar a recuperação de informação em domínios sensíveis e complexos, como os registos criminais da Polícia Judiciária. Ao desenvolver uma API de pesquisa semântica robusta, escalável e tolerante a erros ortográficos, foi possível superar limitações das abordagens tradicionais baseadas em palavras-chave, oferecendo assim uma solução mais alinhada com a complexidade dos dados tratados pela PJ.

A escolha do modelo medcpt-article, após uma avaliação rigorosa de desempenho entre sete alternativas, revelou-se determinante para atingir uma precisão de 95,1%, mesmo que tenho sido originalmente treinado em domínios distintos. Essa capacidade de generalização, aliada a uma arquitetura baseada em microserviços com Docker, React, Flask e FAISS, permitiu construir um sistema modular e eficiente, preparado para evolução contínua.

O projeto seguiu uma abordagem iterativa e prática, enfentrou desafios técnicos como a transição para uma arquitetura containerizada, a definição ideal do tamanho de chunk, e a mitigação de resultados duplicados. Todos esses obstáculos foram ultrapassados com sucesso, consolidando um sistema pronto para possível integração real com sistemas institucionais, como os da Polícia Judiciária, e com potencial de adaptação para outras áreas como saúde ou jornalismo investigativo.

7.2 Principais Conclusões

- A pesquisa semântica demonstrou ser substancialmente mais eficaz do que abordagens tradicionais baseadas em palavras-chave, especialmente em documentos longos e com vocabulário técnico, como os registos criminais.

- A arquitetura modular com buckets independentes garantiu escalabilidade e resiliência, que permitiu a adição dinâmica de novos conjuntos de dados sem comprometer o desempenho do sistema e sem precisar desligar o sistema.

- A utilização de ferramentas como *FAISS*, *Docker*, *Flask*, *React* e *Ollama* revelou-se acertada porque propocionou uma base sólida para o desenvolvimento, integração e expansão da solução.

- A metodologia iterativa adotada foi essencial para a evolução do projeto, permitindo a rápida identificação e resolução de problemas, como o ajuste do tamanho de *chunk*, a filtragem de duplicados e a estabilidade dos serviços em *Docker*.

- A solução desenvolvida provou estar apta não só para a modernização de sistemas judiciais, mas também para ser adaptada a outros contextos onde a recuperação inteligente de informação é crítica.

7.3 Trabalho Futuro

O trabalho futuro prevê o *fine-tuning* do modelo `medcpt_article` em colaboração com a Polícia Judiciária, otimizando-o para dados específicos da PJ. Além disso, planeia-se expandir a API para suportar múltiplos idiomas, melhorar a interface do utilizador com funcionalidades como filtros avançados, otimizar a escalabilidade com técnicas de indexação avançada e explorar funcionalidades como resumo automático de documentos e análise preditiva de padrões criminais, com o objetivo de uma integração eficaz com os sistemas da PJ.

Bibliografia

Bibliografia

[1] Decreto-Lei n.º 35.042, de 20 de Outubro de 1945. Criação da Polícia Judiciária.

Webgrafia

- Polícia Judiciária: `https://www.policiajudiciaria.pt/`
- Direção-Geral da Administração da Justiça: `https://dgaj.justica.gov.pt/`
- Docker: `https://www.docker.com/`
- Python: `https://www.python.org/`
- Flask: `https://flask.palletsprojects.com/`
- React: `https://reactjs.org/`
- Ollama: `https://ollama.com/`
- FAISS: `https://faiss.ai/`
- Git: `https://git-scm.com/`
- GitHub: `https://github.com/`
- Elasticsearch: `https://www.elastic.co/elasticsearch/`
- PyPDF2: `https://pypdf2.readthedocs.io/en/latest/`

Apêndice A

Gestão de Código e Controlo de Versões

O projeto foi gerido com recurso ao sistema de controlo de versões *Git*, com o repositório hospedado na plataforma *GitHub*. Foi utilizado um único ramo principal, denominado `master`, concentrando todo o desenvolvimento da aplicação.

O repositório encontra-se organizado em diretórios específicos, separando os scripts em Python, configurações de *Docker*, ficheiros de composição `docker-compose.yml` e os testes automatizados, promovendo uma estrutura modular e facilmente extensível.

Apêndice B

Documentação

B.1 Guia de Utilização

Este apêndice fornece instruções detalhadas para usar a aplicação, baseadas no README, e uma explicação da interface com imagens ilustrativas.

B.1.1 Instruções do README

Para utilizar a aplicação, siga os passos abaixo:

1. Certifique-se de ter o Docker instalado no seu sistema.
2. Clone o repositório do projeto a partir do GitHub `https://github.com/Rexi10/PRJ_55_49734`.
3. Na pasta raiz do projeto, execute o comando `docker-compose up -d` para iniciar os serviços em *background*.
4. Aceda à interface web no endereço `http://localhost:5000/`.
5. Na interface, selecione os *buckets* que deseja consultar, clicando nas caixas de seleção correspondentes.
6. Insira a sua consulta no campo de pesquisa e defina o número de resultados desejados (k).
7. Clique no botão "Pesquisar" para submeter a consulta. Os resultados serão exibidos com o nome do documento, a similaridade, o *bucket* de origem e o *chunk* relevante.

8. Para descarregar um documento, clique no botão “Descarregar” ao lado do resultado correspondente.

Estas instruções permitem configurar e usar a aplicação de forma eficaz.