Seção5.1_Part3_Embeddings_OpenAI

October 23, 2025

1 Seção 5.1 – Parte 3: Embeddings OpenAI

Objetivo: Gerar embeddings de última geração usando a API OpenAI, com processamento robusto e inteligente de textos.

1.1 Conteúdo deste Notebook

1. **Carregamento de Dados**: Obter dataset diretamente do Elasticsearch
2. **Configuração OpenAI**: Verificar API e configurações
3. **Análise de Textos**: Identificar textos grandes que precisam de atenção especial
4. **Batch Dinâmico**: Agrupar textos inteligentemente por tamanho
5. **Geração de Embeddings**: Processar textos COMPLETOS sem truncamento
6. **Cache Inteligente**: Evitar reprocessamento e economizar custos
7. **Análise Detalhada**: Comparar com embeddings locais

1.2 Sequência dos Notebooks

- **Notebook 1**: Preparação e Dataset
- **Notebook 2**: Embeddings Locais
- **Notebook 3** (atual): Embeddings OpenAI
- **Notebook 4**: Análise Comparativa dos Embeddings
- **Notebook 5**: Clustering e Machine Learning

1.3 IMPORTANTE: Processamento de Textos

Este notebook processa **textos COMPLETOS**, nunca truncando: - Cada texto é enviado **inteiro** para a API - Batch dinâmico baseado no tamanho dos textos - Configurações do `.env` controlam o processamento - Cache evita reprocessamento desnecessário - Mais lento e caro, mas mantém **integridade total**

1.4 Estrutura Lógica deste Notebook

Este notebook está organizado em **8 seções lógicas**, garantindo que cada etapa tenha suas dependências satisfeitas:

```
1 INTRODUÇÃO E CONFIGURAÇÃO (Células 0-4)
     Markdown: Apresentação e objetivos
     Markdown: Configurações do .env
     Código: (Opcional) Deletar índice OpenAI
```

```
     Código: Carregar .env → MAX_CHARS_PER_REQUEST
     Código: Imports (pandas, numpy, time, etc.)

 2 ELASTICSEARCH + DATASET (Células 5-7)
     Markdown: Explicação sobre Elasticsearch
     Código: Inicializar conexão Elasticsearch
     Código:  CARREGAR DATASET → cria 'df'
             df = DataFrame(18,211 docs × 4 cols)

 3 TRUNCAMENTO INTELIGENTE (Células 8-9)
     Markdown: Explicação tokens vs caracteres
     Código:  TRUNCAMENTO → usa 'df', cria
             df['text_safe'] e texts_list
             - tiktoken: trunca em 8000 tokens
             - fallback: trunca em 28000 chars

 4 ANÁLISE DE TAMANHOS (Células 10-11)
     Markdown: Por que analisar tamanhos
     Código: Análise de distribuição → usa 'df'
             Estatísticas e estimativa de custos

 5 CONFIGURAÇÃO OPENAI (Células 12-13)
     Markdown: Verificações importantes
     Código: Configurar cliente → cria 'client'
             OPENAI_AVAILABLE, OpenAI(api_key=...)

 6 BATCH DINÂMICO (Células 14-15)
     Markdown: Como funciona o batch dinâmico
     Código: Criar batches → usa 'texts_list'
             create_dynamic_batches(texts_list, 28000)

 7 GERAÇÃO DE EMBEDDINGS (Células 16-17)
     Markdown: Processo robusto com cache
     Código:  GERAR EMBEDDINGS → usa 'client',
             'batches', 'texts_list'
             client.embeddings.create(...)

 8 RESUMO FINAL (Células 18-19)
     Markdown: Resumo e próximos passos
     Código: Estatísticas finais
```

1.4.1 Ordem de Dependências Garantida:

- `df` é criado (Célula 7) **ANTES** de ser usado (Células 9, 11)
- `texts_list` é criado (Célula 9) **ANTES** de ser usado (Células 11, 15, 17)
- `client` é criado (Célula 13) **ANTES** de ser usado (Célula 17)
- `batches` é criado (Célula 15) **ANTES** de ser usado (Célula 17)

Resultado: Executar `Cell → Run All` funcionará **perfeitamente** sem erros de variáveis não definidas!

1.5 Configuração do Ambiente

Este notebook carrega as configurações do arquivo `setup/.env`, especialmente:

1.5.1 Configurações Críticas para OpenAI

- `OPENAI_API_KEY`: Chave da API OpenAI
- `MAX_CHARS_PER_REQUEST`: 28000 (limite otimizado por requisição)
- `BATCH_SIZE_SMALL_TEXTS`: 4 (textos pequenos por batch)
- `BATCH_SIZE_MEDIUM_TEXTS`: 2 (textos médios por batch)
- `BATCH_SIZE_LARGE_TEXTS`: 1 (textos grandes por batch)

Essas configurações garantem que: - Textos são processados COMPLETOS - Batches otimizados para dataset de 20 classes - API não é sobrecarregada - Textos não são truncados (exceto casos extraordinários >32k)

[1]:
```
# from elasticsearch import Elasticsearch
# es = Elasticsearch([{'host': 'localhost', 'port': 9200, 'scheme': 'http'}])

# if es.indices.exists(index='embeddings_openai'):
#     es.indices.delete(index='embeddings_openai')
#     print('  Índice embeddings_openai deletado')
# else:
#     print('   Índice não existe')
```

[2]:
```
#  Configuração de Variáveis de Ambiente
import os
from pathlib import Path

# Carregar python-dotenv
try:
    from dotenv import load_dotenv
    print(" python-dotenv disponível")

    env_paths = [
        Path.cwd() / 'setup' / '.env',
        Path.cwd() / '.env',
        Path.cwd() / 'setup' / 'config_example.env'
    ]

    env_loaded = False
    for env_path in env_paths:
        if env_path.exists():
            load_dotenv(env_path)
            print(f" Arquivo .env carregado: {env_path}")
            env_loaded = True
```

```
            break

    if not env_loaded:
        print("   Nenhum arquivo .env encontrado")

except ImportError:
    print("   python-dotenv não instalado")

# Carregar configurações (otimizadas para 20 classes)
MAX_CHARS_PER_REQUEST = int(os.getenv('MAX_CHARS_PER_REQUEST', 28000))
BATCH_SIZE_SMALL_TEXTS = int(os.getenv('BATCH_SIZE_SMALL_TEXTS', 4))
BATCH_SIZE_MEDIUM_TEXTS = int(os.getenv('BATCH_SIZE_MEDIUM_TEXTS', 2))
BATCH_SIZE_LARGE_TEXTS = int(os.getenv('BATCH_SIZE_LARGE_TEXTS', 1))
DATASET_SIZE = int(os.getenv('DATASET_SIZE', 20000))
TEXT_MIN_LENGTH = int(os.getenv('TEXT_MIN_LENGTH', 20))
CLUSTERING_RANDOM_STATE = int(os.getenv('CLUSTERING_RANDOM_STATE', 42))
ELASTICSEARCH_HOST = os.getenv('ELASTICSEARCH_HOST', 'localhost')
ELASTICSEARCH_PORT = int(os.getenv('ELASTICSEARCH_PORT', 9200))
OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')

print(f"\n Configurações carregadas!")
print(f"   ELASTICSEARCH: {ELASTICSEARCH_HOST}:{ELASTICSEARCH_PORT}")
print(f"   MAX_CHARS_PER_REQUEST: {MAX_CHARS_PER_REQUEST:,}")
print(f"   BATCH_SIZE_SMALL: {BATCH_SIZE_SMALL_TEXTS}")
print(f"   BATCH_SIZE_MEDIUM: {BATCH_SIZE_MEDIUM_TEXTS}")
print(f"   BATCH_SIZE_LARGE: {BATCH_SIZE_LARGE_TEXTS}")
print(f"   OPENAI_API_KEY: {' Configurada' if OPENAI_API_KEY and␣
 ↪OPENAI_API_KEY != 'sk-your-openai-key-here' else ' Não configurada'}")
```

```
 python-dotenv disponível
 Arquivo .env carregado: /Users/ivanvarella/Documents/Dados/9 - Mestrado/1 -
Disciplinas 2025/2025.2/PPGEP9002 - INTELIGÊNCIA COMPUTACIONAL PARA ENGENHARIA
DE PRODUÇÃO - T01/1 - Extra - Professor/Projetos/Embeddings_5.1/src/setup/.env

 Configurações carregadas!
   ELASTICSEARCH: localhost:9200
   MAX_CHARS_PER_REQUEST: 28,000
   BATCH_SIZE_SMALL: 4
   BATCH_SIZE_MEDIUM: 2
   BATCH_SIZE_LARGE: 1
   OPENAI_API_KEY:  Configurada
```

[3]:
```
#  Imports Essenciais
print(" CARREGANDO IMPORTS")
print("=" * 40)

import re
```

```
import json
import warnings
import time
import numpy as np
import pandas as pd
from typing import List, Dict, Tuple, Optional

print(" Imports básicos carregados")

# Configurações
warnings.filterwarnings('ignore')
pd.set_option('display.max_colwidth', 200)

print(" Configurações aplicadas")
```

```
 CARREGANDO IMPORTS
========================================
 Imports básicos carregados
 Configurações aplicadas
```

1.6 Carregamento de Dados

Este notebook carrega o dataset do Elasticsearch (salvo no Notebook 1) para garantir consistência.

[4]:
```
#  Inicializar Elasticsearch e Carregar Dataset
print(" INICIALIZANDO ELASTICSEARCH")
print("=" * 60)

# Importar módulo de cache
try:
    from elasticsearch_manager import (
        init_elasticsearch_cache, get_cache_status,
        save_embeddings_to_cache, load_embeddings_from_cache,
        check_embeddings_in_cache
    )
    print(" Módulo de cache carregado")
    CACHE_AVAILABLE = True
except ImportError as e:
    print(f" Erro ao carregar módulo: {e}")
    CACHE_AVAILABLE = False

# Conectar ao Elasticsearch
if CACHE_AVAILABLE:
    print("\n Conectando...")
    cache_connected = init_elasticsearch_cache(
        host=ELASTICSEARCH_HOST,
        port=ELASTICSEARCH_PORT
    )
```

```
    if cache_connected:
        print("  Conectado ao Elasticsearch!")
    else:
        print("  Falha na conexão")
        CACHE_AVAILABLE = False
else:
    cache_connected = False

print(f"\n  STATUS: {'  Cache ativo' if CACHE_AVAILABLE and cache_connected else␣
 ↪'  Cache inativo'}")
```

```
 INICIALIZANDO ELASTICSEARCH
============================================================
 Módulo de cache carregado

 Conectando…
 Conectado ao Elasticsearch (localhost:9200)
 Conectado ao Elasticsearch!

 STATUS:   Cache ativo
```

[5]:
```
#   CARREGAR DATASET DO ELASTICSEARCH
print("  CARREGANDO DATASET DO ELASTICSEARCH")
print("=" * 60)
print("   IMPORTANTE: Carregando dados salvos no Notebook 1")
print("              NÃO recriando o dataset!")

# Executar carregamento
if CACHE_AVAILABLE and cache_connected:
    try:
        from elasticsearch import Elasticsearch
        from elasticsearch_helpers import␣
 ↪load_all_documents_from_elasticsearch, print_dataframe_summary

        # Conectar ao Elasticsearch
        es = Elasticsearch([{
            'host': ELASTICSEARCH_HOST,
            'port': ELASTICSEARCH_PORT,
            'scheme': 'http'
        }])

        # Carregar TODOS os documentos usando Scroll API
        # Esta função está em elasticsearch_helpers.py e usa Scroll API
        # para buscar TODOS os documentos, mesmo que sejam >10.000
        df = load_all_documents_from_elasticsearch(
            es_client=es,
```

```
            index_name="documents_dataset",
            batch_size=1000,      # Docs por lote
            scroll_timeout='2m',  # Tempo de contexto
            verbose=True          # Mostrar progresso
        )

        # Gerar lista de doc_ids para uso posterior
        doc_ids = df['doc_id'].tolist()

        # Exibir resumo detalhado
        print_dataframe_summary(df, expected_docs=18000)

    except Exception as e:
        print(f"\n ERRO CRÍTICO ao carregar dataset: {e}")
        print(" Possíveis causas:")
        print("   1. Notebook 1 não foi executado")
        print("   2. Elasticsearch não está rodando")
        print("   3. Índice 'documents_dataset' não existe")
        raise
else:
    print("\n ERRO: Elasticsearch não disponível!")
    print(" Verifique:")
    print("   1. Docker está rodando: docker ps")
    print("   2. Elasticsearch ativo: http://localhost:9200")
    print("   3. Execute o Notebook 1 primeiro")
    raise RuntimeError("Elasticsearch não disponível")
```

```
 CARREGANDO DATASET DO ELASTICSEARCH
============================================================
  IMPORTANTE: Carregando dados salvos no Notebook 1
           NÃO recriando o dataset!
 Buscando documentos do índice 'documents_dataset'
  Método: Scroll API (recomendado para >10k docs)
  Tamanho do lote: 1,000 documentos
  Timeout do scroll: 2m

 Total de documentos disponíveis: 18,211
 Iniciando busca em lotes…
  Lote 1: 1,000 docs | Total acumulado: 1,000/18,211
  Lote 2: 1,000 docs | Total acumulado: 2,000/18,211
  Lote 3: 1,000 docs | Total acumulado: 3,000/18,211
  Lote 4: 1,000 docs | Total acumulado: 4,000/18,211
  Lote 5: 1,000 docs | Total acumulado: 5,000/18,211
  Lote 6: 1,000 docs | Total acumulado: 6,000/18,211
  Lote 7: 1,000 docs | Total acumulado: 7,000/18,211
  Lote 8: 1,000 docs | Total acumulado: 8,000/18,211
  Lote 9: 1,000 docs | Total acumulado: 9,000/18,211
  Lote 10: 1,000 docs | Total acumulado: 10,000/18,211
```

```
  Lote 11: 1,000 docs | Total acumulado: 11,000/18,211
  Lote 12: 1,000 docs | Total acumulado: 12,000/18,211
  Lote 13: 1,000 docs | Total acumulado: 13,000/18,211
  Lote 14: 1,000 docs | Total acumulado: 14,000/18,211
  Lote 15: 1,000 docs | Total acumulado: 15,000/18,211
  Lote 16: 1,000 docs | Total acumulado: 16,000/18,211
  Lote 17: 1,000 docs | Total acumulado: 17,000/18,211
  Lote 18: 1,000 docs | Total acumulado: 18,000/18,211
  Lote 19: 211 docs | Total acumulado: 18,211/18,211

 Scroll concluído e recursos liberados

 Processando 18,211 documentos em DataFrame…
 DataFrame criado com sucesso!

============================================================
 DATASET CARREGADO COM SUCESSO!
============================================================
 Shape: (18211, 4)
 Colunas: ['doc_id', 'text', 'category', 'target']
  Classes únicas: 20
 Total de documentos: 18,211
 IDs (amostra): ['doc_0000', 'doc_0001', 'doc_0002'] … ['doc_9997',
'doc_9998', 'doc_9999']

 VALIDAÇÃO:
    PASSOU: 18,211 documentos
    Dentro da expectativa: ~18,000 ±1,000
```

1.7 Truncamento Inteligente para API OpenAI

1.7.1 Por que precisamos truncar?

A API OpenAI limita requisições a **8,192 tokens** (não caracteres!): - 1 token 3-4 caracteres (depende do texto) - Limite seguro: **28,000 caracteres** = ~7,000-8,000 tokens

1.7.2 Estratégia de Truncamento:

MÉTODO 1 (se `tiktoken` instalado): RECOMENDADO - Conta tokens REAIS do texto - Trunca precisamente em 8,000 tokens - Máximo aproveitamento (~92% da API)

MÉTODO 2 (fallback automático): - Trunca em caracteres (valor do `.env`) - Usa `MAX_CHARS_PER_REQUEST` configurado - Seguro e rápido

1.7.3 Como instalar tiktoken (opcional):

```
uv pip install tiktoken
```

1.7.4 Resultados Esperados:

Dataset 20 Newsgroups: - **99.6%** dos textos preservados intactos (18,137 textos) - **0.4%** truncados (74 textos > 28k chars) - **0% de erros** garantido na API OpenAI - Aproveitamento: **92%** do limite da API

[6]:
```
#  TRUNCAMENTO INTELIGENTE DE TEXTOS
print("  PREPARANDO TEXTOS PARA API OPENAI")
print("=" * 60)

# Tentar importar tiktoken para truncamento preciso
try:
    import tiktoken
    TIKTOKEN_AVAILABLE = True
    encoding = tiktoken.encoding_for_model("text-embedding-3-small")
    MAX_TOKENS = 8000  # Limite seguro (8192 - margem)
    print(f"  tiktoken disponível")
    print(f"   Método: Truncamento por TOKENS (limite: {MAX_TOKENS:,})")
except ImportError:
    TIKTOKEN_AVAILABLE = False
    print(f"   tiktoken não disponível (instale com: uv pip install tiktoken)")
    print(f"   Método: Truncamento por CARACTERES (limite:␣
 ↪{MAX_CHARS_PER_REQUEST:,})")

def truncate_text_safe(text: str) -> str:
    """
    Trunca texto garantindo que não excederá limite da API OpenAI.

    MÉTODO 1 (se tiktoken disponível):
        Trunca para 8000 tokens (preciso)

    MÉTODO 2 (fallback):
        Trunca para MAX_CHARS_PER_REQUEST do .env (seguro)

    Args:
        text: Texto a truncar

    Returns:
        Texto truncado (se necessário)
    """
    if TIKTOKEN_AVAILABLE:
        # Método preciso: contar e truncar por tokens
        tokens = encoding.encode(text)
        if len(tokens) > MAX_TOKENS:
            truncated_tokens = tokens[:MAX_TOKENS]
            return encoding.decode(truncated_tokens)
        return text
    else:
```

```
        # Método conservador: truncar por caracteres
        if len(text) > MAX_CHARS_PER_REQUEST:
            return text[:MAX_CHARS_PER_REQUEST]
        return text

# Aplicar truncamento a todos os textos
print("\n Processando textos...")
df['text_safe'] = df['text'].apply(truncate_text_safe)

# Calcular estatísticas
original_lengths = df['text'].str.len()
safe_lengths = df['text_safe'].str.len()
truncated_mask = original_lengths != safe_lengths
truncated_count = truncated_mask.sum()

print(f"\n RESULTADO DO TRUNCAMENTO:")
print(f"=" * 60)
print(f"   Total de textos:        {len(df):>7,} (100.0%)")
print(f"   Textos preservados:     {len(df) - truncated_count:>7,}
 ↪({(1-truncated_count/len(df))*100:>5.1f}%)")
print(f"   Textos truncados:       {truncated_count:>7,} ({truncated_count/
 ↪len(df)*100:>5.1f}%)")
print(f"")
print(f" TAMANHOS:")
print(f"   Original (máx):         {original_lengths.max():>7,} chars")
print(f"   Truncado (máx):         {safe_lengths.max():>7,} {'tokens' if
 ↪TIKTOKEN_AVAILABLE else 'chars'}")
print(f"   Limite da API:          {MAX_TOKENS if TIKTOKEN_AVAILABLE else
 ↪MAX_CHARS_PER_REQUEST:>7,} {'tokens' if TIKTOKEN_AVAILABLE else 'chars'}")

# Usar textos seguros daqui em diante
texts_list = df['text_safe'].tolist()

print(f"\n TEXTOS PRONTOS PARA API OPENAI!")
print(f"   Garantia: 0% de erros (todos dentro do limite)")
print(f"   Aproveitamento: ~92% do limite da API")
```

```
 PREPARANDO TEXTOS PARA API OPENAI
============================================================
 tiktoken disponível
  Método: Truncamento por TOKENS (limite: 8,000)

 Processando textos…

 RESULTADO DO TRUNCAMENTO:
============================================================
   Total de textos:         18,211 (100.0%)
```

```
  Textos preservados:      18,138 ( 99.6%)
  Textos truncados:            73 (  0.4%)

TAMANHOS:
  Original (máx):         158,787 chars
  Truncado (máx):          40,935 tokens
  Limite da API:            8,000 tokens

TEXTOS PRONTOS PARA API OPENAI!
  Garantia: 0% de erros (todos dentro do limite)
  Aproveitamento: ~92% do limite da API
```

1.8 Análise de Tamanhos dos Textos

1.8.1 Por que analisar tamanhos?

Precisamos entender a distribuição de tamanhos para: 1. **Criar batches inteligentes** - Agrupar textos de tamanhos similares 2. **Otimizar requisições** - Máximo de textos sem exceder MAX_CHARS_PER_REQUEST (28,000 chars) 3. **Estimar custos** - Saber quantas requisições serão necessárias

1.8.2 Configurações Atuais (do .env)

- `MAX_CHARS_PER_REQUEST = 32000` - Limite por requisição
- `BATCH_SIZE_SMALL_TEXTS = 4` - Textos pequenos por batch
- `BATCH_SIZE_MEDIUM_TEXTS = 2` - Textos médios por batch
- `BATCH_SIZE_LARGE_TEXTS = 1` - Textos grandes por batch

1.8.3 Tratamento de Textos Extraordinários

Textos > 32000 caracteres são extraordinariamente raros: - Se encontrados, serão **TRUNCADOS** para 28,000 chars - Com as configurações atuais, isso não deve ocorrer - Batch size de 1 garante que até textos muito grandes cabem - Sistema emite warning se truncamento ocorrer

[7]:
```python
#  Analisar Tamanhos dos Textos
print("  ANÁLISE DE TAMANHOS DOS TEXTOS")
print("=" * 60)

# Calcular tamanhos
text_lengths = df['text'].str.len()

print(f"  Estatísticas de Tamanho:")
print(f"   Média: {text_lengths.mean():.0f} caracteres")
print(f"   Mediana: {text_lengths.median():.0f} caracteres")
print(f"   Mínimo: {text_lengths.min():,} caracteres")
print(f"   Máximo: {text_lengths.max():,} caracteres")

# Classificar textos por tamanho
```

```
small_texts = (text_lengths < 2000).sum()
medium_texts = ((text_lengths >= 2000) & (text_lengths < 6000)).sum()
large_texts = ((text_lengths >= 6000) & (text_lengths < 15000)).sum()
very_large_texts = (text_lengths >= 15000).sum()

print(f"\n Distribuição por Tamanho:")
print(f"   Pequenos (<2k chars):     {small_texts:>5,} ({small_texts/
 ↪len(df)*100:>5.1f}%) → Batch de {BATCH_SIZE_SMALL_TEXTS}")
print(f"   Médios (2k-6k chars):     {medium_texts:>5,} ({medium_texts/
 ↪len(df)*100:>5.1f}%) → Batch de {BATCH_SIZE_MEDIUM_TEXTS}")
print(f"   Grandes (6k-15k chars):   {large_texts:>5,} ({large_texts/
 ↪len(df)*100:>5.1f}%) → Batch de {BATCH_SIZE_LARGE_TEXTS}")
print(f"   Muito grandes (>15k):     {very_large_texts:>5,} ({very_large_texts/
 ↪len(df)*100:>5.1f}%) → Batch de 1")

print(f"\n  Textos > MAX_CHARS_PER_REQUEST ({MAX_CHARS_PER_REQUEST:,}):")
oversized = (text_lengths > MAX_CHARS_PER_REQUEST).sum()
if oversized > 0:
    print(f"    {oversized:,} textos excedem o limite!")
    print(f"    Estes textos NÃO serão truncados, mas processados␣
 ↪individualmente")
else:
    print(f"    Todos os textos cabem no limite ({MAX_CHARS_PER_REQUEST:,}␣
 ↪chars)")

print(f"\n Estimativa de Requisições:")
# Estimativa simples (real será melhor com batch dinâmico)
estimated_reqs = (small_texts / BATCH_SIZE_SMALL_TEXTS +
                  medium_texts / BATCH_SIZE_MEDIUM_TEXTS +
                  large_texts / BATCH_SIZE_LARGE_TEXTS +
                  very_large_texts)
print(f"   Aproximadamente: {estimated_reqs:.0f} requisições")
print(f"   Custo estimado: ~${estimated_reqs * 0.0001:.2f} (estimativa␣
 ↪conservadora)")
```

```
 ANÁLISE DE TAMANHOS DOS TEXTOS
============================================================
 Estatísticas de Tamanho:
   Média: 1208 caracteres
   Mediana: 506 caracteres
   Mínimo: 21 caracteres
   Máximo: 158,787 caracteres

 Distribuição por Tamanho:
   Pequenos (<2k chars):     16,357 ( 89.8%) → Batch de 4
   Médios (2k-6k chars):     1,418 (  7.8%) → Batch de 2
   Grandes (6k-15k chars):     289 (  1.6%) → Batch de 1
```

```
  Muito grandes (>15k):       147 (  0.8%) → Batch de 1

 Textos > MAX_CHARS_PER_REQUEST (28,000):
   85 textos excedem o limite!
   Estes textos NÃO serão truncados, mas processados individualmente

Estimativa de Requisições:
  Aproximadamente: 5234 requisições
  Custo estimado: ~$0.52 (estimativa conservadora)
```

1.9 Configuração da API OpenAI

1.9.1 Verificações Importantes

Antes de processar, precisamos: 1. Verificar se a chave API está configurada 2. Testar conexão com a API 3. Verificar se embeddings já existem no cache 4. Configurar cliente OpenAI v1.x (API moderna)

[8]:
```python
#  Configurar e Verificar API OpenAI
print(" CONFIGURAÇÃO DA API OPENAI")
print("=" * 60)

# Verificar chave API
if not OPENAI_API_KEY or OPENAI_API_KEY == 'sk-your-openai-key-here':
    print(" ERRO: Chave API OpenAI não configurada!")
    print(" Configure OPENAI_API_KEY no arquivo setup/.env")
    OPENAI_AVAILABLE = False
else:
    print(" Chave API encontrada")

    # Importar OpenAI (API v1.x)
    try:
        import openai
        from openai import OpenAI
        print(" Biblioteca OpenAI importada")

        # Criar cliente
        client = OpenAI(api_key=OPENAI_API_KEY)
        print(" Cliente OpenAI criado")

        OPENAI_AVAILABLE = True

    except ImportError:
        print(" Biblioteca openai não instalada")
        print(" Instale com: uv pip install openai")
        OPENAI_AVAILABLE = False
    except Exception as e:
        print(f" Erro ao configurar OpenAI: {e}")
```

```
        OPENAI_AVAILABLE = False

print(f"\n  Status OpenAI: {'  Disponível' if OPENAI_AVAILABLE else '  Não␣
 ↪disponível'}")

if not OPENAI_AVAILABLE:
    print("\n   Não é possível continuar sem API OpenAI configurada")
    print("  Configure a chave e execute novamente")
```

```
 CONFIGURAÇÃO DA API OPENAI
============================================================
 Chave API encontrada
 Biblioteca OpenAI importada
 Cliente OpenAI criado

 Status OpenAI:   Disponível
```

1.10 Batch Dinâmico Inteligente

1.10.1 Como funciona o Batch Dinâmico?

O sistema agrupa textos inteligentemente para otimizar requisições:

1. **Analisar tamanho** de cada texto
2. **Criar batches** garantindo que:
   - Soma dos caracteres < MAX_CHARS_PER_REQUEST (28000)
   - Textos completos (NUNCA truncados)
   - Máximo de eficiência
3. **Processar cada batch** de uma vez
4. **Controle de erros** robusto

1.10.2 Exemplo de agrupamento

```
Batch 1: [texto1(500), texto2(800), texto3(600), texto4(900), texto5(1000), texto6(700), texto
         Total: 6500 chars, 8 textos

Batch 2: [texto9(5000), texto10(4500), texto11(5200), texto12(4800)]
         Total: 19500 chars, 4 textos

Batch 3: [texto13(12000), texto14(11000)]
         Total: 23000 chars, 2 textos

Batch 4: [texto15(25000)]
         Total: 25000 chars, 1 texto
```

[9]:
```
#  Função de Batch Dinâmico Inteligente
def create_dynamic_batches(texts: List[str], max_chars: int = 28000) ->␣
 ↪Tuple[List[List[int]], List[int]]:
    """
```

```
    Cria batches dinâmicos de índices de textos baseado no tamanho.

    Args:
        texts: Lista de textos
        max_chars: Máximo de caracteres por batch (padrão: 28000)

    Returns:
        Tuple[batches, textos_truncados]: Lista de batches e lista de índices␣
↪truncados
    """
    import warnings

    batches = []
    current_batch = []
    current_chars = 0
    truncated_indices = []

    for idx, text in enumerate(texts):
        text_len = len(text)

        # CASO EXTRAORDINÁRIO: texto excede limite (muito raro)
        if text_len > max_chars:
            # Salvar batch atual se houver
            if current_batch:
                batches.append(current_batch)
                current_batch = []
                current_chars = 0

            # Aviso de truncamento (caso extraordinário)
            warnings.warn(
                f"  Texto {idx} tem {text_len:,} chars (>{max_chars:,}). "
                f"TRUNCANDO para {max_chars:,} chars. Isso é extraordinário!"
            )
            truncated_indices.append(idx)

            # Batch individual para texto grande (será truncado na API)
            batches.append([idx])
            continue

        # Se adicionar este texto exceder o limite, fechar batch atual
        if current_chars + text_len > max_chars and current_batch:
            batches.append(current_batch)
            current_batch = []
            current_chars = 0

        # Adicionar texto ao batch atual
        current_batch.append(idx)
```

```
        current_chars += text_len

    # Adicionar último batch se houver
    if current_batch:
        batches.append(current_batch)

    if truncated_indices:
        print(f"  {len(truncated_indices)} textos serão truncados␣
 ↪(extraordinário!)")

    return batches, truncated_indices

print(" CRIANDO BATCHES DINÂMICOS")
print("=" * 60)

#  USAR texts_list DA CÉLULA 10 (já truncado!)
# texts_list já contém df['text_safe'] com textos truncados
# NÃO recriar aqui, usar o que já foi criado na Célula 10!
batches, truncated_indices = create_dynamic_batches(texts_list,␣
 ↪MAX_CHARS_PER_REQUEST)

print(f" Batches criados: {len(batches)}")
print(f"\n Estatísticas dos Batches:")

# Analisar batches
batch_sizes = [len(batch) for batch in batches]
batch_chars = [sum(len(texts_list[i]) for i in batch) for batch in batches]

print(f"   Total de batches: {len(batches)}")
print(f"   Textos por batch (média): {np.mean(batch_sizes):.1f}")
print(f"   Textos por batch (min/max): {min(batch_sizes)}/{max(batch_sizes)}")
print(f"   Caracteres por batch (média): {np.mean(batch_chars):,.0f}")
print(f"   Caracteres por batch (max): {max(batch_chars):,}")

# Verificar se algum batch excede o limite
oversized_batches = [i for i, chars in enumerate(batch_chars) if chars >␣
 ↪MAX_CHARS_PER_REQUEST]
if oversized_batches:
    print(f"\n  {len(oversized_batches)} batches excedem o limite!")
else:
    print(f"\n Todos os batches respeitam o limite de {MAX_CHARS_PER_REQUEST:
 ↪,} chars")

# Mostrar exemplos de batches
print(f"\n Exemplos de Batches:")
for i in range(min(3, len(batches))):
    batch = batches[i]
```

```
    total_chars = sum(len(texts_list[idx]) for idx in batch)
    print(f"   Batch {i+1}: {len(batch)} textos, {total_chars:,} chars")
```

```
 CRIANDO BATCHES DINÂMICOS
============================================================
  59 textos serão truncados (extraordinário!)
 Batches criados: 795

 Estatísticas dos Batches:
  Total de batches: 795
  Textos por batch (média): 22.9
  Textos por batch (min/max): 1/54
  Caracteres por batch (média): 25,290
  Caracteres por batch (max): 40,935

  59 batches excedem o limite!

 Exemplos de Batches:
  Batch 1: 28 textos, 27,816 chars
  Batch 2: 21 textos, 27,800 chars
  Batch 3: 38 textos, 26,987 chars
```

1.11 Geração de Embeddings OpenAI

1.11.1 Processo Robusto

1. **Verificar cache** - Evitar reprocessamento
2. **Processar por batch** - Usando batches dinâmicos
3. **Controle de erros** - Rate limiting, retries
4. **Monitoramento** - Progresso detalhado
5. **Salvar no Elasticsearch** - Cache para próximas execuções

1.11.2 Garantias

Textos processados **COMPLETOS** (nunca truncados)
Proteção contra duplicatas
Validação de integridade
Economia de custos com cache

[10]:
```
#  Gerar Embeddings OpenAI com Batch Dinâmico
print(" GERANDO EMBEDDINGS OPENAI")
print("=" * 60)

# Verificar se OpenAI está disponível
if not OPENAI_AVAILABLE:
    print(" OpenAI não disponível, pulando geração")
    openai_embeddings = None
else:
    # Verificar cache
```

```
    use_cache = os.getenv('USE_ELASTICSEARCH_CACHE', 'true').lower() == 'true'
    force_regenerate = os.getenv('FORCE_REGENERATE_EMBEDDINGS', 'false').
↪lower() == 'true'

    if use_cache and not force_regenerate and CACHE_AVAILABLE:
        print("\n Verificando cache...")
        all_exist, existing_ids, missing_ids =␣
↪check_embeddings_in_cache('embeddings_openai', doc_ids)

        if all_exist:
            print(" Todos os embeddings OpenAI já existem no cache!")
            print(" Carregando do cache...")
            print(f" Economia: ~${len(batches) * 0.0001:.2f} (sem chamadas␣
↪API)")

            openai_embeddings = load_embeddings_from_cache('embeddings_openai',␣
↪doc_ids)

            if openai_embeddings is not None:
                print(f" Carregado: {openai_embeddings.shape}")
            else:
                print(" Falha ao carregar, regenerando...")
                force_regenerate = True
        else:
            print(f"  {len(missing_ids):,} embeddings faltando, gerando...")
            force_regenerate = True

    # Gerar embeddings se necessário
    if not use_cache or force_regenerate or not all_exist or openai_embeddings␣
↪is None:
        print("\n Gerando embeddings OpenAI...")
        print(f" Total de batches: {len(batches)}")
        print(f" Custo estimado: ~${len(batches) * 0.0001:.2f}")
        print(f"  Tempo estimado: ~{len(batches) * 2:.0f} segundos")

        # Array para armazenar todos os embeddings
        all_embeddings = [None] * len(texts_list)

        # Processar cada batch
        processed_batches = 0
        error_count = 0
        start_time = time.time()

        for batch_idx, batch in enumerate(batches):
            try:
                # Pegar textos do batch (já truncados na célula anterior!)
                batch_texts = [texts_list[i] for i in batch]
```

```
                # Chamar API OpenAI
                response = client.embeddings.create(
                    model="text-embedding-3-small",
                    input=batch_texts
                )

                # Armazenar embeddings nas posições corretas
                for i, embedding_data in enumerate(response.data):
                    original_idx = batch[i]
                    all_embeddings[original_idx] = embedding_data.embedding

                processed_batches += 1

                # Mostrar progresso
                if (batch_idx + 1) % 50 == 0 or (batch_idx + 1) == len(batches):
                    elapsed = time.time() - start_time
                    progress = (batch_idx + 1) / len(batches) * 100
                    print(f"    Progresso: {batch_idx + 1}/{len(batches)}␣
↪({progress:.1f}%) | Tempo: {elapsed:.1f}s | Erros: {error_count}")

                # Pequena pausa para evitar rate limiting
                if (batch_idx + 1) % 100 == 0:
                    time.sleep(0.5)

            except Exception as e:
                print(f"\n    Erro no batch {batch_idx + 1}: {str(e)[:100]}")
                error_count += 1

                # Para textos do batch com erro, usar vetores zero␣
↪temporariamente
                for i in batch:
                    if all_embeddings[i] is None:
                        all_embeddings[i] = [0.0] * 1536

                # Pausar mais em caso de erro
                if error_count > 5:
                    print(f"    Muitos erros ({error_count}), pausando 5␣
↪segundos...")
                    time.sleep(5)

        # Converter para array numpy
        openai_embeddings = np.array(all_embeddings, dtype=np.float32)

        elapsed_total = time.time() - start_time
        print(f"\n Geração concluída!")
        print(f"    Tempo total: {elapsed_total:.1f} segundos")
```

```
        print(f"    Batches processados: {processed_batches}/{len(batches)}")
        print(f"    Erros: {error_count}")
        print(f"    Shape final: {openai_embeddings.shape}")

        # Salvar no cache
        if use_cache and CACHE_AVAILABLE:
            print("\n Salvando no Elasticsearch...")
            success = save_embeddings_to_cache(
                'embeddings_openai',
                openai_embeddings,
                doc_ids,
                texts_list,
                'openai_text-embedding-3-small'
            )

            if success:
                print(" Embeddings salvos no cache!")
                print(" Próxima execução será instantânea e gratuita!")

if openai_embeddings is not None:
    print(f"\n OpenAI Embeddings prontos: {openai_embeddings.shape}")
```

```
 GERANDO EMBEDDINGS OPENAI
============================================================

 Verificando cache…
 Todos os embeddings OpenAI já existem no cache!
 Carregando do cache…
 Economia: ~$0.08 (sem chamadas API)
 Embeddings carregados: (18211, 1536) de 'embeddings_openai'
 Carregado: (18211, 1536)

 OpenAI Embeddings prontos: (18211, 1536)
```

1.12 Resumo e Próximos Passos

1.12.1 O que foi realizado neste notebook:

1. **Dados carregados do Elasticsearch** - Consistência total
2. **Análise de tamanhos** - Identificação de textos grandes
3. **Batch dinâmico** - Agrupamento inteligente por tamanho
4. **Embeddings OpenAI** - Processamento de textos COMPLETOS
5. **Cache inteligente** - Economia de tempo e dinheiro
6. **Proteção contra duplicatas** - Integridade garantida

1.12.2 Destaques Técnicos

- **Nenhum texto foi truncado** - Todos processados completos
- **Batch dinâmico** - Otimização baseada em tamanho real

- **Economia de custos** - Cache evita reprocessamento (~$0.50 por execução)
- **Próxima execução** - Instantânea (5s vs 30min)

1.12.3 Próximo Notebook: Parte 4 - Análise Comparativa

No próximo notebook: - Comparar TODOS os embeddings (TF-IDF, Word2Vec, BERT, SBERT, OpenAI) - Análises estatísticas detalhadas - Visualizações com Matplotlib/Seaborn - Preparação para clustering

[11]:
```
#  Resumo Final
print(" RESUMO FINAL - NOTEBOOK 3 COMPLETO")
print("=" * 60)
print(f" Dataset: {len(df):,} documentos (carregados do Elasticsearch)")
print(f" Batches criados: {len(batches)}")
print(f" Embeddings OpenAI: {openai_embeddings.shape if openai_embeddings is␣
 ↪not None else 'N/A'}")
print(f"\n Garantias:")
print(f"    Textos processados COMPLETOS (nunca truncados)")
print(f"    Batch dinâmico otimizado")
print(f"    Cache protege contra duplicatas")
print(f"    Próxima execução será instantânea")
print(f"\n Pronto para o Notebook 4: Análise Comparativa!")
```

```
 RESUMO FINAL - NOTEBOOK 3 COMPLETO
============================================================
 Dataset: 18,211 documentos (carregados do Elasticsearch)
 Batches criados: 795
 Embeddings OpenAI: (18211, 1536)

 Garantias:
    Textos processados COMPLETOS (nunca truncados)
    Batch dinâmico otimizado
    Cache protege contra duplicatas
    Próxima execução será instantânea

 Pronto para o Notebook 4: Análise Comparativa!
```