

Código

Primeira Etapa - Leitura

```
# primeiro passo, devemos utilizar da biblioteca pandas para que
# possamos ler esse csv

import pandas as pd

df = pd.read_csv("reclamacoes_clientes.csv")

df.head(2)
```

Segunda Etapa - tratamento


```
import spacy # importa a biblioteca spaCy para processamento de linguagem

nlp = spacy.load("pt_core_news_sm") # carrega o modelo de linguagem para

def preprocess(text):
    if not isinstance(text, str): # verifica se o input é uma string, senão retorna
        return ""
    doc = nlp(text.lower()) # converte o texto para minúsculas e processa com
    # cria uma lista de lemas (forma base das palavras) filtrando tokens que são
    tokens = [token.lemma_ for token in doc if token.is_alpha and not token.is_stop]
    print(tokens) # imprime os tokens lematizados para conferência
    return " ".join(tokens) # retorna os tokens unidos em uma string novamente

# aplica a função preprocess em cada texto da coluna 'resposta_aberta' e cria
df["processed_text"] = df["reclamacao"].apply(preprocess)

# imprime as primeiras linhas das colunas originais e processadas para comp
print(df[["reclamacao", "processed_text"]].head())
```

`text.lower()` → tudo em minúsculas = consistência e redução de dimensionalidade do vocabulário 

✓ Por que usar `lower()` (converter para minúsculas)?

1. Evita duplicidade no vocabulário

Palavras como `"Banco"`, `"BANCO"` e `"banco"` são **a mesma palavra** para nós, mas para o modelo seriam **três tokens diferentes** se não forem convertidas para minúsculas.

Sem `lower()` :

```
text
CopiarEditar
['Banco', 'banco', 'BANCO'] → 3 palavras distintas
```

Com `lower()` :

```
text
CopiarEditar
['banco', 'banco', 'banco'] → 1 única palavra
```

2. Reduz o tamanho do vocabulário

- Modelos de linguagem funcionam melhor com um vocabulário menor e mais uniforme.
- Isso **melhora desempenho, economiza memória e aumenta a precisão** dos embeddings.

3. Evita viés causado por capitalização

- Nomes próprios ou palavras em início de frase (que começam com maiúscula) podem parecer importantes só por estarem com letra maiúscula — o que nem sempre é verdade.
- Converter tudo para minúsculo **normaliza o contexto**.
- `token.is_alpha` → remove números, pontuação, emojis etc. ✓
- `not token.is_stop` → remove stopwords ("de", "a"...). ✓
- `token.lemma_` → mantém o **lema** da palavra (forma base, como "fazer" em vez de "fazendo") ✓

Terceira Etapa - Chunks

```
# iremos utilizar os chunks pois os textos estao muito grandes, dessa forma p
from langchain.text_splitter import CharacterTextSplitter

texto_longo = " ".join(df["processed_text"].tolist())

# com o separator conseguimos separar corretamente esses chunks
splitter = CharacterTextSplitter(chunk_size=70, chunk_overlap=20, separator=

chunks = splitter.split_text(texto_longo)

print(chunks)
```

Contexto geral

Você tem um texto muito longo (no seu caso, concatenado a partir da coluna `"processed_text"` de um DataFrame `df`) e quer processá-lo em partes menores, chamadas **chunks**.

Por que usar chunks?

- **Limite de tokens:** APIs de modelos de linguagem (como GPTs) normalmente têm limite de tokens que podem ser enviados numa única requisição. Se o texto for muito grande, você ultrapassa esse limite.
- **Performance:** Processar o texto em pedaços menores pode ser mais eficiente, evita sobrecarga, e pode melhorar a precisão e organização dos dados, principalmente em tarefas como embeddings, sumarização, ou busca semântica.
- **Economia de tokens:** Reduz o custo de uso da API porque você processa o texto em partes, evitando enviar tudo junto repetidamente.

O que o código faz?

1. **Concatenação do texto longo:**

```
python
CopiarEditar
texto_longo = " ".join(df["processed_text"].tolist())
```

Você pega todos os textos processados do seu DataFrame `df`, transforma numa lista e junta numa única string grande, separando cada texto com espaço.

1. Configuração do splitter:

```
python
CopiarEditar
splitter = CharacterTextSplitter(chunk_size=70, chunk_overlap=20, separator=" ")
```

- `chunk_size=70` : cada pedaço terá até 70 caracteres.
- `chunk_overlap=20` : pedaços consecutivos terão uma sobreposição de 20 caracteres. Isso ajuda a manter contexto entre os chunks, pois evita quebras bruscas.
- `separator=" "` : o splitter vai tentar dividir o texto nas quebras de espaço para não cortar palavras no meio.

1. Divisão do texto em chunks:

```
python
CopiarEditar
chunks = splitter.split_text(texto_longo)
```

Aqui o texto longo é quebrado em vários pedaços de até 70 caracteres, com sobreposição e respeitando os espaços.

1. Impressão dos chunks:

```
python
CopiarEditar
```

```
print(chunks)
```

Você vê a lista de pedaços gerados.

Quarta Etapa - Transformação para embeddings

```
# agora precisamos fazer o embeddamento para que transformemos em um vetor
# para armazenar em um banco de dados vetorizado

from sentence_transformers import SentenceTransformer

model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")

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

embeddings_chunks = model.encode(chunks)

print(f"Gerados {len(embeddings_chunks)} embeddings para chunks, dimensão: {embeddings_chunks[0].shape}")
```

- **O que são embeddings?**

Embeddings são representações numéricas de textos (ou outras informações) em um espaço vetorial. Cada texto é transformado em um vetor de números que captura seu significado semântico.

- **Por que transformar chunks em embeddings?**

Depois de dividir seu texto em pedaços menores (chunks), você quer representá-los numericamente para poder fazer buscas, comparações e análises eficientes. Computadores entendem números, então converter texto para vetores é essencial para trabalhar com linguagem natural.

- **O que faz o código?**

- `model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")` : carrega um modelo pré-treinado que converte textos em vetores, suportando múltiplos idiomas.
- `embeddings_chunks = model.encode(chunks)` : transforma cada chunk em um vetor (embedding). O resultado é uma lista de vetores, um para cada chunk.

- Cada vetor tem uma dimensão fixa (por exemplo, 384 números), que codifica a informação semântica daquele pedaço de texto.
- **Para que usar esses vetores?**

Esses vetores podem ser armazenados em bancos de dados vetorizados para buscas rápidas, similaridade entre textos, recuperação de informações e outros processos que dependem de entender o conteúdo além das palavras exatas.

Quinta Etapa - Armazenamento no chromaDB

```
# agora devemos armazenar em um banco de dados para que consigamos re
# semantica

import chromadb
from chromadb.config import Settings

client = chromadb.Client(Settings())

collection = client.get_or_create_collection(name="reclamacoes")

ids = [str(i) for i in range(len(chunks))]

metadatas = [{"Source": "reclamacao", "chunk_index": i} for i in range(len(chunks))]

collection.add(
    ids=ids,
    documents=chunks,
    embeddings=embeddings_chunks,
    metadatas=metadatas
)
```

Passo a passo:

1. Importação e Configuração do Cliente ChromaDB

```
python
CopiarEditar
import chromadb
from chromadb.config import Settings

client = chromadb.Client(Settings())
```

- Você está criando uma conexão com o banco de dados vetorial *ChromaDB*.
- `Settings()` usa configurações padrão para inicializar o cliente.

1. Criar ou obter uma coleção no banco

```
python
CopiarEditar
collection = client.get_or_create_collection(name="reclamacoes")
```

- Uma *coleção* funciona como uma "tabela" onde você armazena os dados.
- Se a coleção chamada `"reclamacoes"` não existir, ela será criada.

1. Preparar os dados para armazenamento

```
python
CopiarEditar
ids = [str(i) for i in range(len(chunks))]
metadata = [{"Source": "reclamacao", "chunk_index": i} for i in range(len(chunks))]
```

- Cada chunk terá um `id` único, aqui apenas números sequenciais convertidos para string.
- Você também adiciona *metadata* para cada chunk, com informações úteis (por exemplo, a fonte do texto e o índice do chunk), que podem ser usadas para filtrar ou entender o contexto dos dados depois.

1. Adicionar dados na coleção

```
python
CopiarEditar
collection.add(
    ids=ids,
    documents=chunks,
    embeddings=embeddings_chunks,
    metadatas=metadatas
)
```

- Aqui você envia para o banco:
 - Os `ids` para identificar cada chunk,
 - Os textos originais (`documents`),
 - Os vetores numéricos (`embeddings`) que representam o conteúdo,
 - E as informações adicionais (`metadatas`).

Por que fazer isso?

- **Armazenar embeddings em um banco vetorial** permite que você faça buscas semânticas, ou seja, busque textos que tenham significado parecido, mesmo que não contenham as mesmas palavras.
- Usar metadados ajuda a organizar e filtrar resultados depois da busca.
- **documents**: são os textos originais, os pedaços (*chunks*) do conteúdo que você quer indexar e buscar depois.
- **metadatas**: são informações adicionais relacionadas a cada texto, usadas para identificar, categorizar ou filtrar resultados na busca (por exemplo, a origem do texto e a posição do chunk no documento).

Sexta Etapa - Consulta

```
# agora ja podemos realizar uma consulta nesse nosso banco de dados vetor
```



```

query_text = "As entregas são feitas dentro do prazo?"

query_embedding = model.encode([query_text][0])

results = collection.query(
    query_embeddings=[query_embedding],
    n_results=3,
    include=["documents", "distances", "metadatas"]
)

print("Top 3 resultados mais similares")

for doc, dist, meta in zip(results["documents"][0], results["distances"][0], results["metadatas"][0]):
    print(f"Distância: {dist:.4f} | Meta: {meta} | Texto: {doc}\n")

```

O que o código faz:

1. **Define uma consulta** em texto: "As entregas são feitas dentro do prazo?" .
2. **Gera o embedding** desse texto usando o mesmo modelo de transformação que usamos para os chunks.
3. **Consulta o banco vetorial** (ChromaDB) buscando os 3 documentos mais similares ao embedding da consulta.
4. **Exibe os 3 resultados** mais próximos, mostrando:
 - A distância (similaridade) entre a consulta e o documento;
 - Os metadados associados ao documento;
 - O texto original do chunk.

Por que isso importa:

A consulta é transformada em vetor para comparar semanticamente com os textos armazenados. A busca retorna os textos mais relevantes ao significado da pergunta, não apenas correspondência literal de palavras.