# Tokens, tokenização e embeddings

Escola de Verão 2026

Alexandre Teles <alexandre.teles@inctdd.org>

# Índice

# Introdução

Modelos de linguagem trabalham com texto, mas não entendem o texto da mesma forma que humanos. Para que modelos de linguagem possam processar o texto, ele precisa ser convertido em uma forma que o modelo possa entender. Uma vez que modelos de linguagem não "leem" texto como humanos, eles precisam de uma representação numérica do texto.

Para transformar inguagem em números, usamos três ideias-chave:

1. Tokens: as unidades básicas do texto.
2. Tokenização: o processo de dividir o texto em tokens.
3. Embeddings: representações numéricas com significado.

Entender esses conceitos é essencial para compreender como LLMs funcionam por dentro.

# Tokens

## O que são tokens?

Tokens são as unidades básicas processadas por modelos de linguagem. Podem ser palavras, partes de palavras, símbolos ou até mesmo caracteres individuais, dependendo do tokenizador utilizado.

É importante notar que tokens não são necessariamente iguais a palavras. Por exemplo, a palavra "incrível" pode ser dividida em dois tokens: `in` e `crivel`, dependendo do tokenizador. Modelos **não veem texto**, apenas **IDs de tokens**.

Tokens nos permitem:

1. Lidar com **vocabulários extensos**
2. Tratar **palavras raras ou novas**
3. Manter **consistência** na representação do texto
4. Padronizar o tamanho das entradas

# Tokens

## 🟦 Como escolher tokens?

- **Tamanho do vocabulário**
  - pequeno → sequências longas
  - grande → mais memória e parâmetros
- **Nível de granularidade**
  - caracteres ↔ palavras ↔ subpalavras
- **Eficiência computacional**
  - menos tokens = processamento mais rápido
- **Generalização**
  - subpalavras lidam melhor com palavras novas
- **Custo**
  - treinamento, memória, inferência

*(Tokenização é sempre um compromisso, não uma escolha perfeita.)*

# Tokenização

Tokenização é o processo de **converter texto bruto em uma sequência de tokens** que um modelo de linguagem consegue processar. Como modelos operam apenas sobre números, cada token é posteriormente **mapeado para um identificador inteiro**.

Esse processo define **como o modelo "enxerga" a linguagem**. A mesma frase pode ser tokenizada de formas diferentes, dependendo das regras adotadas (palavras, subpalavras, caracteres). Por isso, a tokenização influencia diretamente **eficiência, generalização e custo computacional**.

Na prática, tokenizar envolve etapas como **normalização do texto**, divisão em unidades menores e mapeamento para IDs em um vocabulário.

# Tokenização

## Exemplo conceitual

```python
from typing import List, Dict

def tokenize(text: str, vocab: Dict[str, int]) -> List[int]:
    """
    Converte texto em uma sequência de IDs de tokens.
    """
    # Normalização simples
    text = text.lower()

    # Divisão em tokens (exemplo didático)
    tokens: List[str] = text.split()

    # Mapeamento de tokens para IDs
    token_ids: List[int] = [vocab[token] for token in tokens]

    return token_ids


vocab = {"olá": 1, "mundo": 2}

tokenize("Olá mundo", vocab)  # → [1, 2]
```

*(Tokenizadores reais usam subpalavras e métodos estatísticos, mas a ideia central é a mesma.)*

# Tokenização

Quase todos os tokenizadores modernos (BPE, Unigram, WordPiece, etc.) partem da mesma ideia estatística:

> Aprender, a partir das frequências no corpus, uma forma de segmentar o texto que seja estatisticamente eficiente e reutilizável.

ou, em outras palavras:

> Dividir o texto em partes que aparecem com frequência suficiente para valer a pena tratá-las como unidades.

Tokenizadores aprendem tokens a partir da distribuição estatística do corpus, portanto:

- Sequências **frequentes** viram tokens únicos
- Sequências **raras** são decompostas em partes menores
- Isso está intimamente ligado a **compressão de dados

# Tokenizadores

## BPE (Byte Pair Encoding)

1. Começa com caracteres
2. Junta pares mais frequentes repetidamente

Aqui a intuição é a seguinte: **se algo aparece muito, vale a pena virar um símbolo próprio.**

De forma geral, dado um corpus `D` representado como uma sequência de **caracteres**:

$$D = (s_1, s_2, ..., s_N)$$

Em cada iteração, escolhe-se o par adjacente mais frequente:

$$(a^*, b^*) = \arg\max_{a,b} \text{freq}(a, b)$$

Cria-se um novo tken `c` a partir desse par:

$$c = (a^*, b^*)$$

E substitui-se todas as ocorrências de `(a*, b*)` por `c` no corpus `D`. O processo é repetido até atingir um limite de merges ou de vocabulário.

# Tokenizadores

## Unigram (LM)

Ideia: ter um vocabulário `V` de tokens candidatos (subpalavras) e um modelo que atribui uma probabilidade a cada token:

$$p(t) \text{ for } t \in V$$

*(Lê-se: para todo token `t` em `V`, define-se uma probabilidade `p(t)`.)*

**Inferência (tokenizar um texto `x`)**: escolhe a segmentação mais provável.

$$s^* = \arg\max_s p(s \mid x)$$

*(Lê-se: `s*` é a segmentação `s` que maximiza `p(s | x)`.)*

Uma forma de pensar: a probabilidade do texto soma sobre tokenizações possíveis.

$$p(x) = \sum_s p(s \mid x)$$

*(Lê-se: `p(x)` é a soma, para toda segmentação `s` possível de `x`, de `p(s | x)`.)*

# Treinando um tokenizador

Um dos principais hiperparâmetros ao treinar um tokenizador é o tamanho do vocabulário. Um vocabulário maior resulta em menos tokens por texto, mas aumenta o custo computacional e a complexidade do modelo. Já um vocabulário menor reduz o custo, mas pode levar a sequências de tokens mais longas e menos eficientes.

Artigos recentes sugerem utilizar a proximidade da frequência dos tokens à lei de Zipf como critério de pontuação automática uma vez que a linguagem natural segue distribuição Zipfiana: palavras frequentes são muito comuns, palavras raras são extremamente raras `(f(r) ∝ 1/r^α)`.

Vocabulários que produzem tokens com distribuição Zipfiana vão apresentar melhor desempenho downstream porque:

- **Alinhamento estrutural:** tokenização reflete a estrutura estatística natural do corpus
- **Vocabulário muito pequeno:** força sobre-segmentação, achata a curva, modelo precisa aprender padrões longos para reconstruir palavras básicas
- **Vocabulário muito grande:** muitos tokens raros sub-treinados, cauda pesada, parâmetros desperdiçados

**Zipf ótimo = granularidade ideal:** unidades frequentes mapeiam para poucos tokens frequentes; unidades raras são composições de tokens comuns (onde possível).

# Treinando um tokenizador

**Clique aqui para abrir o notebook de treinamento de tokenizador** (*https://colab.research.google.com/drive/1qplja9Bm8Vc9JoNmiwKGK8hz1rfmguBy?usp=sharing*)