

## 一. LLM Tokenization 分词化

在大语言模型（LLM）的应用中，令牌（Token）和嵌入向量（Embedding）是两个核心概念。如果不了解 Token 和 Embedding，就不能清楚地了解 LLM 是如何工作的。语言模型处理的是称为 Token 的小块文本。语言模型计算处理语言的方式是将 token 转换为称为 embedding 的数字表示。

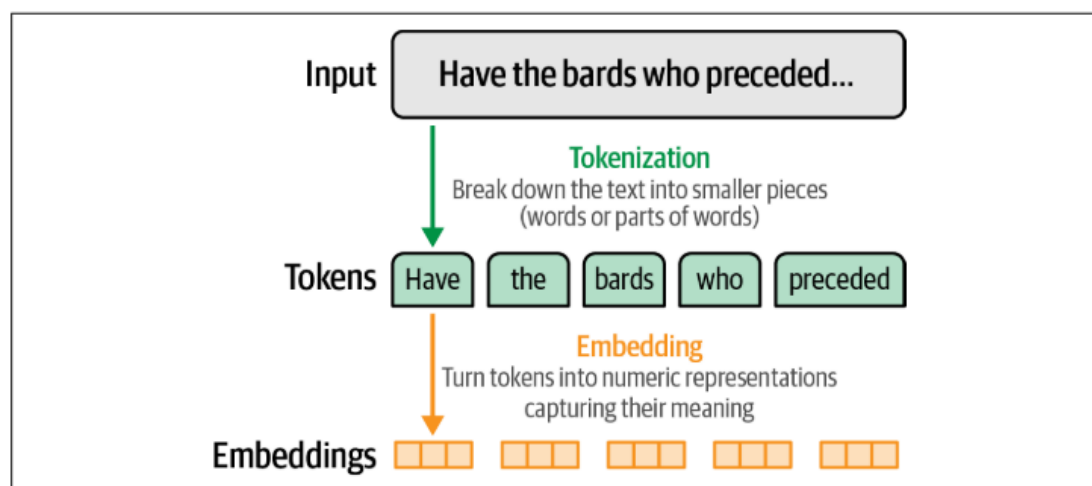


Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

在这一章中，我们将更仔细地研究什么是 Token，以及驱动大语言模型的 tokenization 的方法。然后我们将深入研究在 LLM 出现之前著名的 word2vec 嵌入方法。最后，我们将从 token embedding 深入到 sentence embedding 或 document embedding，也就是整个句子或整个文档都可以用单个向量来表示，这与本书第二部分实现语义搜索和主题建模有关。

### 1. 分词器如何准备提供给语言模型的输入

在使用 ChatGPT 等大模型时你会发现，模型并不是一次性输出所有字，而是一个一个字生成，实际上模型是一次生成一个 Token。Token 不仅是模型的输出方式，也是模型看待处理语言的方式。发送到模型的文本首先被分解为 token。

在提示被输入到大语言模型之前，输入文本首先通过一个将其分解为多个 token 的 Tokenizer(分词器)。图 2-3 展示了 GPT4 的分词器，如果将文本提供给它，它会输出颜色不同的 token，每个 token 都以不同的颜色显示。

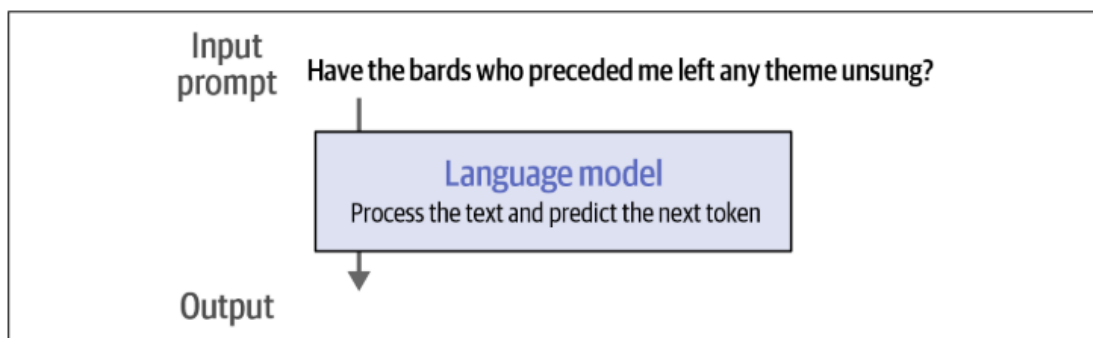


Figure 2-2. High-level view of a language model and its input prompt.

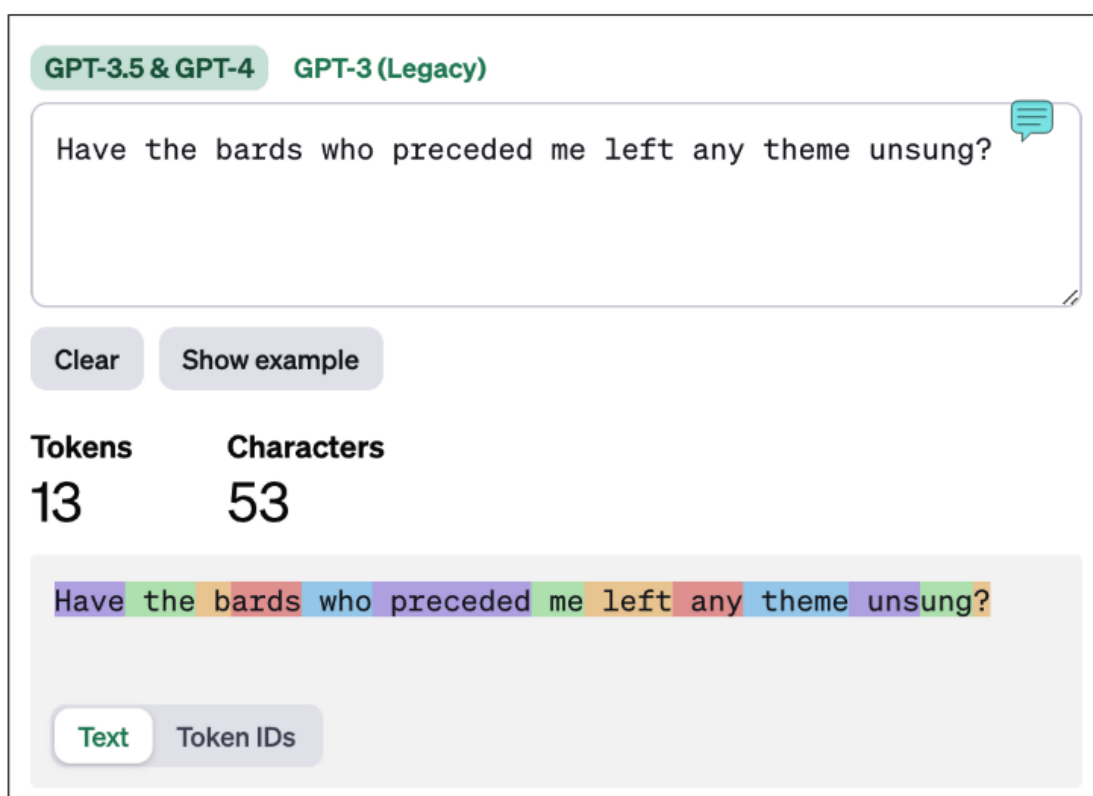


Figure 2-3. A tokenizer breaks down text into words or parts of words before the model processes the text. It does so according to a specific method and training procedure (from <https://oreil.ly/ovUWO>).

接下来是一段代码示例，我们将下载 LLM 并了解如何在使用 LLM 生成文本之前对输入进行分词化。

## 2. 下载并运行 LLM

```
%capture
!pip install transformers>=4.41.2 sentence-transformers>=3.0.1 gensim>=4.3.2 scikit-learn>=1.5.0 accelerate>=0.31.0
```

注意，需要先升级 Numpy 才能成功调用 transformers 库

```
# 升级numpy到最新版本
!pip install --upgrade numpy
```

就像第一章中所做的，先下载模型和它对应的分词器。

下载模型：

```
from transformers import AutoModelForCausalLM, AutoTokenizer

/usr/local/lib/python3.12/dist-packages/sklearn/utils/_param_validation.py:14: UserWarning: A NumPy version >=1.22.4 and
from scipy.sparse import csr_matrix, issparse

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct", #指定要加载的预训练模型
    device_map="cuda", #指定模型运行的设备，将模型加载到GPU上运行（需要NVIDIA GPU和CUDA）
    torch_dtype="auto", #自动选择数值精度
    trust_remote_code=False,
)

/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens).
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
config.json: 100% ██████████ 967/967 [00:00<00:00, 108kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: ██████████ 16.5k/? [00:00<00:00, 1.27MB/s]
Fetching 2 files: 100% ██████████ 2/2 [02:58<00:00, 178.02s/it]
model-00001-of-00002.safetensors: 100% ██████████ 4.97G/4.97G [02:57<00:00, 33.7MB/s]
model-00002-of-00002.safetensors: 100% ██████████ 2.67G/2.67G [02:22<00:00, 10.3MB/s]
Loading checkpoint shards: 100% ██████████ 2/2 [00:27<00:00, 13.08s/it]
generation_config.json: 100% ██████████ 181/181 [00:00<00:00, 21.0kB/s]
```

下载分词器：

```
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

tokenizer_config.json: ██████████ 3.44k/? [00:00<00:00, 335kB/s]
tokenizer.model: 100% ██████████ 500k/500k [00:00<00:00, 1.25MB/s]
tokenizer.json: ██████████ 1.94M/? [00:00<00:00, 25.8MB/s]
added_tokens.json: 100% ██████████ 306/306 [00:00<00:00, 23.9kB/s]
special_tokens_map.json: 100% ██████████ 599/599 [00:00<00:00, 54.7kB/s]
```

然后，我们可以使用模型进行输出。我们首先声明提示符，然后对其进行分词化，随后将这些 token 传递给模型，模型生成输出。在本例中，我们要求模型仅生成 20 个 token：

```
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.<|assistant|>"
#要求写一封向Sarah道歉的邮件，解释园艺事故的发生

# Tokenize the input prompt
input_ids = tokenizer(prompt, #将文本转换为模型能理解的数字token序列
                      return_tensors="pt" #返回PyTorch张量 (tensor)
                      ).input_ids.to("cuda") #提取分词后的数字ID序列

# Generate the text
generation_output = model.generate( #调用模型的生成方法
    input_ids=input_ids, #提供分词后的输入
    max_new_tokens=20 #输出20个token
)

# Print the output
print(tokenizer.decode #将数字token序列转换回可读文本
      (generation_output[0]))
```

The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a consequence, you may observe the following behavior: the model will generate tokens until it reaches the end of the sequence (EOS token) which is not what you want, as the model will keep generating tokens. Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.<|assistant|> Subject: Heartfelt Apologies

输出：Subject 后是模型生成的文本。

```
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.<|assistant|> Subject: Heartfelt Apologies for the Gardening Mishap
```

通过以上代码可以看到，模型实际上没有收到完整的提示，而是 tokenizer 先处理输入提示，并在变量 input\_ids 中返回提示分词后生成的 token，模型将这些 token 用作输入。

打印 input\_ids 查看其中的内容：

```
print(input_ids)
```

```
tensor([[14350,   385,  4876, 27746,  5281,   304, 19235,   363,   278, 25305,
         293, 16423,   292,   286,   728,   481, 29889, 12027,  7420,   920,
         372,  9559, 29889, 32001]], device='cuda:0')
```

这揭示了 LLM 所处理的输入形式，如图 2-4 所示，这些输入实际上是一系列整数序列。每个整数都对应着特定 token 的唯一标识符（ID），这些 token 可能是字符、单词或单词的组成部分。这些 ID 指向分词器内部的一个词表，该词表包含了分词器所识别的所有 token，每个 token 都有一个唯一的 ID。

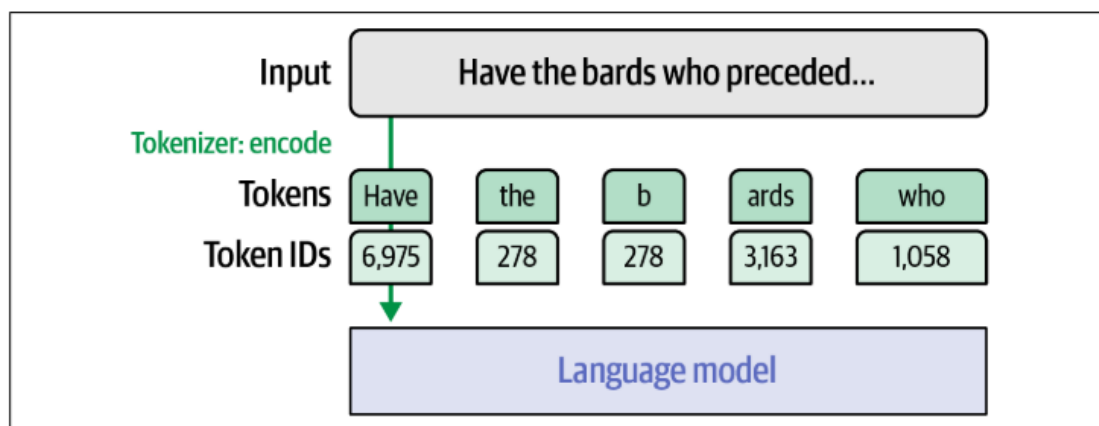


Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

我们可以将 `input_ids` 解码，转换回人类可以阅读的文本：

```
for id in input_ids[0]:  
    print(tokenizer.decode(id))
```

Write  
an  
email  
apolog  
izing  
to  
Sarah  
for  
the  
trag  
ic  
garden  
ing  
m  
ish  
ap  
.  
Exp  
lain  
how  
it  
happened  
.  
<|assistant|>

注意，有些 token 是完整的单词，有些 token 是单词的一部分。空格没有自己的 token。有些 token 开头有一个特殊的隐藏字符，表示它们与前一个 token 有联系，在文本中表现为属于同一个单词。如果 Token 开头没有特殊的字符，说明这是一个单词的开头或一个完整的单词。

我们可以通过打印 `generation_output` 来检查模型生成的 Token:

```
generation_output
tensor([[14350, 385, 4876, 27746, 5281, 304, 19235, 363, 278, 25305,
        293, 16423, 292, 286, 728, 481, 29889, 12027, 7420, 920,
        372, 9559, 29889, 32001, 3323, 622, 29901, 17778, 29888, 2152,
        6225, 11763, 363, 278, 19906, 292, 341, 728, 481, 13,
        13, 13, 29928, 799]], device='cuda:0')
```

蓝底是模型生成的内容: 3323 是“Sub”, 622 是“ject”, 它们一起组成了“Subject”。29901 代表冒号。与输入一样, 我们需要输出端的分词器将 token ID 转换成人类可读的文本。可以向分词器传递单个 token ID 也可以传入列表。

```
print(tokenizer.decode(3323))
print(tokenizer.decode(622))
print(tokenizer.decode([3323, 622]))
print(tokenizer.decode(29901))

Sub
ject
Subject
:
```

### 3. 分词器如何分解文本

决定分词器如何分解输入提示的因素主要有三个。

第一, 在模型设计阶段, 创建者会选择一种分词方法。主流方法包括字节对编码 (byte pair encoding, BPE) (GPT 系列模型广泛采用) 和 WordPiece (BERT 模型使用)。这些方法的共同点在于它们都致力于优化表示文本数据集的高效标记集合, 但实现路径有所不同。

第二, 选定方法后, 需要制定分词器的设计决策, 例如词汇表的大小和特殊标记的使用。更多细节详见“5. 比较经过训练的 LLM 分词器”

第三, 分词器需在特定数据集上进行训练, 以建立能最优表示该数据集的词汇表。即使采用相同方法和参数, 在英文文本数据集上训练的分词器也会与在代码数据集或多语言文本数据集上训练的分词器存在差异。

分词器除用于将输入文本处理为语言模型可读格式 (token) 外, 还作用于语言模型的输出端: 将生成的标记 ID 转换为对应的输出词或标记, 如图 2-5 所示。

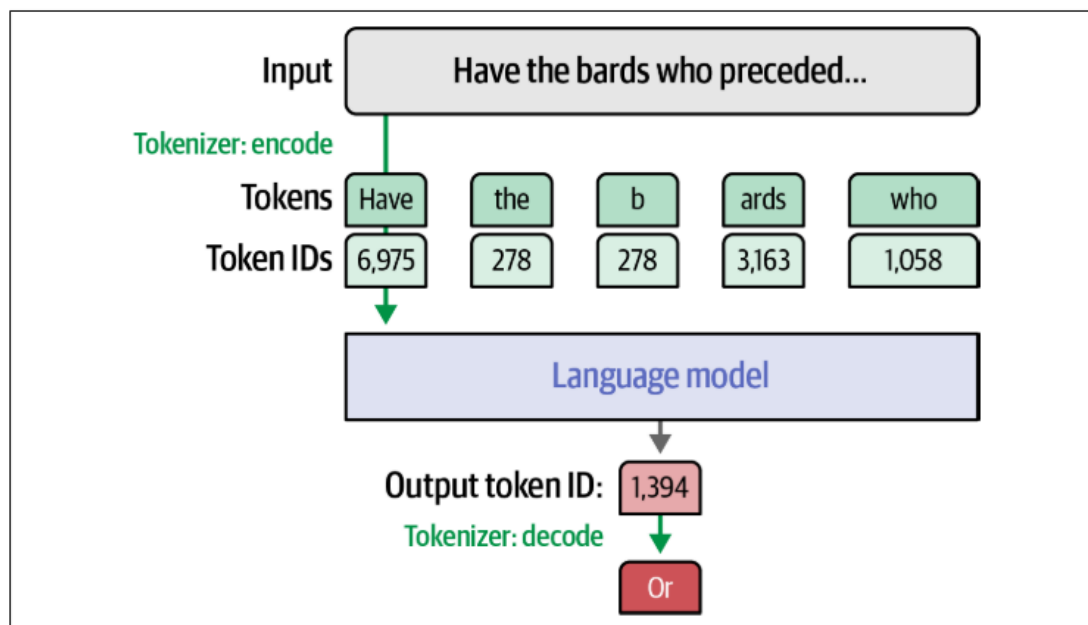


Figure 2-5. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

4. word(单词) VS subword(子词) VS character(字符) VS byte tokens(字节词元)

上面我们讨论的分词化方案称为子词分词化,这是最常用的分词化方案但不是唯一的,以下是四种值得注意的分词化方法:

#### ① Word tokens 单词级 Token

这种方法在早期技术(如 word2vec)中较为常见,但在自然语言处理(NLP)领域的应用正逐渐减少。不过,其实用性促使它在 NLP 之外的其他场景(如推荐系统)中仍被采用,最后一节推荐系统 Recommendation Systems 将展开说明。

但是,当训练后的分词器遇到未收录的新词时可能无法处理。同时,这会导致词汇表中存在大量仅有细微差异的标记(例如 apology、apologize、apologetic、apologist)。Subword tokens 很好地解决了这个问题。

#### ② Subword tokens 子词级 Token

子词分词(subword tokenization)拆分词根与词缀,例如用“apolog”作为基础标记,搭配通用后缀标记(如-y、-ize、-etic、-ist)组合成词。这种方案既提升了词汇表的表达能力,又能通过常见字符片段重构新词。另一核心价值在于:通过将新词拆解为更小的字符单元(这些单元通常已存在于词汇表中),实现对新词的表示能力。

### ③ Character tokens 字符级 Token

回溯至最基础的字母单元。虽然这种表示方式更易于分词，但会增加建模难度。在子词分词模型中，“play”可能被表示为单个标记，而字符级标记模型则需要先拼写出“p-l-a-y”的字符序列，再对其上下文进行建模。

但这种分词方法也有弊端：token 数量的增加，在一定的上下文长度前提下容纳更少的文本。相较于字符标记，子词标记能在 Transformer 模型有限的上下文长度内容纳更多文本。例如对于一个上下文长度为 1,024 的模型，使用子词分词处理的文本量可达字符标记的三倍左右（子词标记的平均长度通常为每个标记含 3 个字符）。

### ④ Byte tokens 字节级 Token

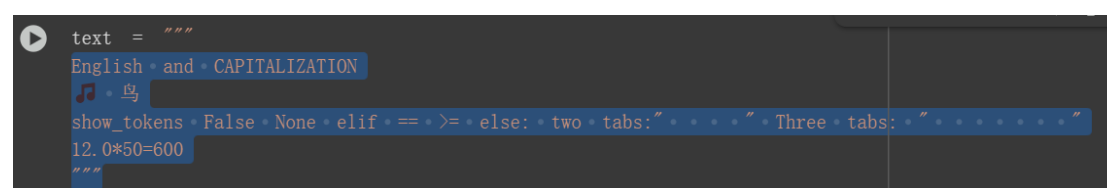
另一种分词方法将标记分解为表示 Unicode 字符的独立字节。如论文《“CANINE: Pre-training an efficient tokenization-free encoder for language representation》中提出的方法（被称为“无分词编码”）。《ByT5: Towards a token-free future with pre-trained byte-to-byte models》也采用了这种方案并证明在多语言场景下具有竞争力。

另外值得注意的是，某些子词分词器会将字节作为最终备用标记纳入词汇表，当遇到无法表示的字符时可回退至字节单元。例如 GPT-2 和 RoBERTa 的分词器就采用这种策略。

## 5. 比较经过训练的 LLM 分词器

如前所述，决定分词器中所含分词的三大关键因素在于：分词方法的选择、初始化分词器时采用的参数与特殊标记，以及分词器的训练数据集。接下来我们将通过对比多个经实际训练的分词器，具体观察这些选择如何影响其行为特征。此对比不仅将揭示新式分词器为提升模型性能而做出的行为调整，还将展现在代码生成等专业领域模型中，定制化分词器往往具有不可或缺的作用。

我们将使用不同的分词器来编码以下这段文本：



```
text = """
English and CAPITALIZATION
🎵 鸟
show_tokens=False=None elif == >= else: two tabs: " " " Three tabs: " " " "
12.0*50=600
"""
```

这段文本中包含了不同种类的 Tokens：大写字母、英文以外的语言、emojis、使



用关键词和空格（用于缩进）编写的代码、数字和特殊分词（通常不是用来表示文本的，它们通常指示文本开始或文本结束）。

我们用出现时间不同的几个分词器对文本进行分词化,然后使用以下函数使用颜色背景色打印每个标记:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

colors_list = [ #RGB编码, 六个颜色循环打印
    '102;194;165', '252;141;98', '141;160;203',
    '231;138;195', '166;216;84', '255;217;47'
]

def show_tokens(sentence, tokenizer_name):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    token_ids = tokenizer(sentence).input_ids
    for idx, t in enumerate(token_ids):
        print(
            # idx % len(colors_list)是取余数,
            # 不断输出0, 1, 2, 3, 4, 5, 循环取色
            f'\x1b[0;30;48;2;{colors_list[idx % len(colors_list)]}m' +
            tokenizer.decode(t) +
            '\x1b[0m',
            end=' ' #每个token之后输出一个空格
        )
```

## ①BERT base model (uncased) (2018)

google-bert/bert-base-uncased • Hugging Face

分词方法: WordPiece; 词表大小: 30522;

有五种特殊分词：

unk token [UNK] 分词器无法提供确定编码的未知 token;

sep\_token [SEP] 分隔符，用来分隔两个不同的句子或文本。针对需要提供给模型两段文本的任务，例如分类、语义相似度判断和自然语言推理：

pad\_token[PAD] 填充标记，由于输入给模型的序列是一定长度的，用来填充输入中未使用的位置：

cls token[CLS] 分类任务的特殊分类 token, 详见第四章;

mask token[MASK] 掩码 token, 用于在训练过程中遮掩 Token。

```
> show_tokens(text, "bert-base-uncased")
```

tokenizer\_config.json: 100% ██████████ 48.0/48.0 [00:00<00:00, 4.55kB/s]

config.json: 100% ██████████ 570/570 [00:00<00:00, 67.1kB/s]

vocab.txt: 100% ██████████ 232k/232k [00:00<00:00, 17.6MB/s]

tokenizer.json: 100% ██████████ 466k/466k [00:00<00:00, 6.50MB/s]

[tokens] english and capitalization UNK UNK show token size none 91 ## # class two tab ## target tab ## f1 i1 50 = 500 [SSA]

BERT 主要有两种发行形式：cased（保留大写）和 Uncased（所有大写先转换为

小写), Uncased 更流行。

观察上面的打印结果,有四点值得注意:不会识别换行符;所有字母都是小写的;##字符用于指示此 token 是连接到其前面的 token 的部分令牌,表示属于单词的一个部分。这也是一种指示空格位置的方法,因为前面没有##的 token 意味着前面有一个空格;无法识别 emoji 和中文。

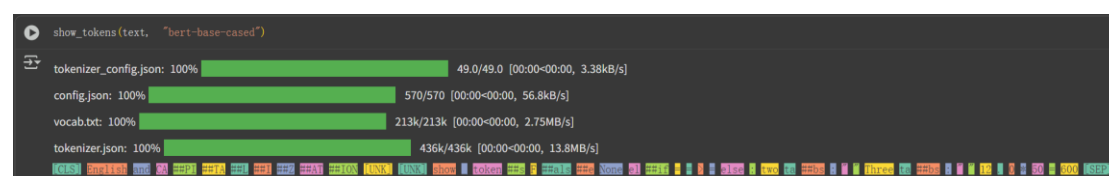
## ②BERT base model(cased) (2018)

<https://huggingface.co/google-bert/bert-base-cased>

分词方法: WordPiece; 词表大小: 2996;

特殊分词和 uncased 一样。与 uncased 的不同之处在于大写 Token。

注意 CAPITALIZATION 被八个 Token 表示; Uncased 和 cased 的开头和结尾都有 [CLS]和[SEP]。



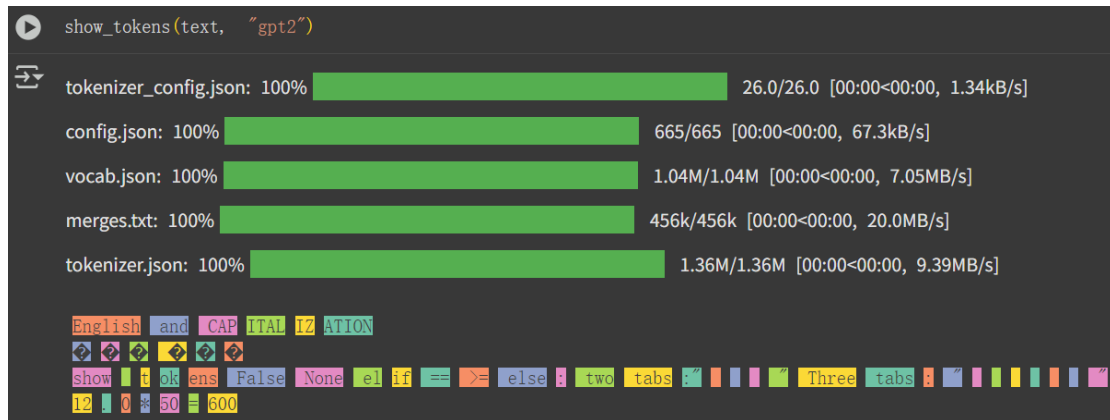
## ③GPT-2 (2019)

<https://huggingface.co/openai-community/gpt2>

分词方法: 字节对编码; 详见论文《Neural machine translation of rare words with subword units》 词表大小: 50257

特殊分词: <endoftext>

有四点值得注意:换行符可以识别;保留了大写字母;emoji 和“鸟”有多个 tokens 表示,虽然这些 tokens 被打印成“?”,但它们实际上代表了不同的 Tokens。例如 emoji 被分解为 ID 为 8582、236、113 的 token,分词器成功地重建出了原始字符。我们可以打印 `tokenizer.decode([8582, 236, 113])` 来验证这一点,输出结果正是代表音符的 emoji; tabs 有对应的 Tokens 进行表示。

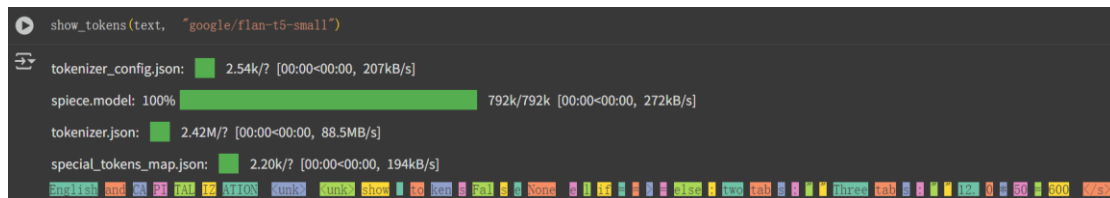


### ⑤ Flan-T5(2022)

使用名为 SentencePiess 的分词器，详见论文《SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing》，它支持 BPE 和 unigram language model（详见论文《Subword regularization: Improving neural network translation models with multiple subword candidates》）

词表大小：32100 特殊分词：unk\_token[unk], pad\_token[pad]

有两点需要注意：换行符和空格没有 Token，因此模型难以处理代码；不能识别 emoji 和“鸟”。



### ⑥ GPT-4(2023)

分词方法：BPE；词表大小：略高于十万；

特殊分词：<|endoftext|>和<|fim\_prefix|><|fim\_middle|><|fim\_suffix|>

<|fim\_prefix|>：前缀标记。代表输入文本的第一部分。

<|fim\_suffix|>：后缀标记。代表输入文本的最后部分。

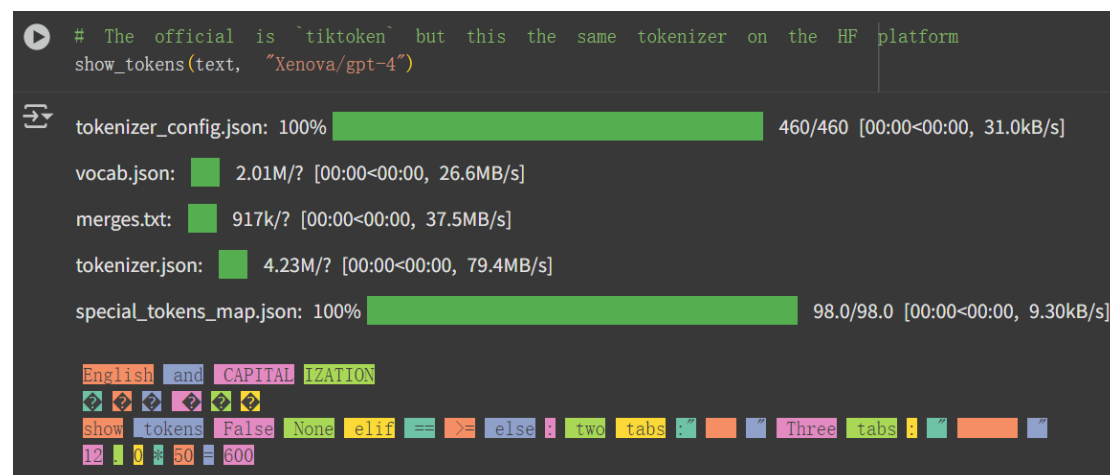
<|fim\_middle|>：中间标记。代表需要模型生成填充的中间部分。

更多细节详见论文《Efficient training of language models to fill in the middle》。

GPT-4 分词器很像 GPT-2 的分词器，但也有不同：将多个空格表示为单个 Token，最多可包含 83 个空格；Python 的关键字 elif 有自己的 Token；使用更

少的 token 来表示大多数单词，如“CAPITALIZATION”用两个 Token 表示，“tokens”用一个 token 表示。

```
# The official is `tiktoken` but this the same tokenizer on the HF platform
show_tokens(text, "Xenova/gpt-4")
```



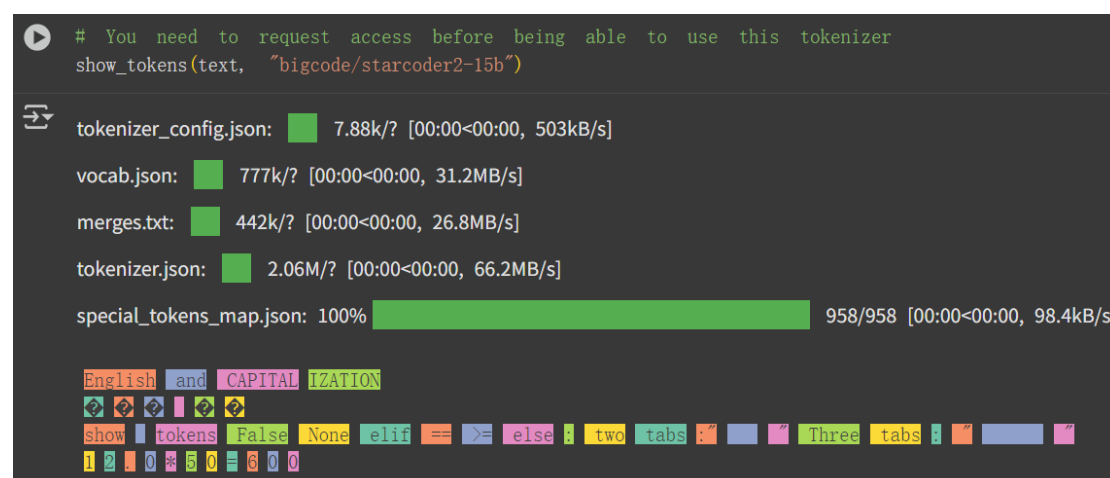
## ⑦ StarCoder2 (2024)

StarCoder2 是一个 150 亿参数的模型，专注于生成代码，详见论文《StarCoder 2 and the stack v2: The next generation》。第一代 StarCoder 详见论文《StarCoder: May the source be with you!》

分词方法：BPE；词表大小：49152；特殊标记：<|endoftext|>，<|fim\_prefix|><|fim\_middle|><|fim\_suffix|><|fim\_pad|>，<filename><reponame><gh\_stars>

需要注意的是：与 GPT-4 类似，将空格序列编码成一个 token；与之前所见过的所有分词器不同，数字是按单个数字来生成 token，例如 600 变成了“6”“0”“0”，这是因为这种方式可以更好地表示数字和数学。

```
# You need to request access before being able to use this tokenizer
show_tokens(text, "bigcode/starcoder2-15b")
```



## ⑧ Galactica

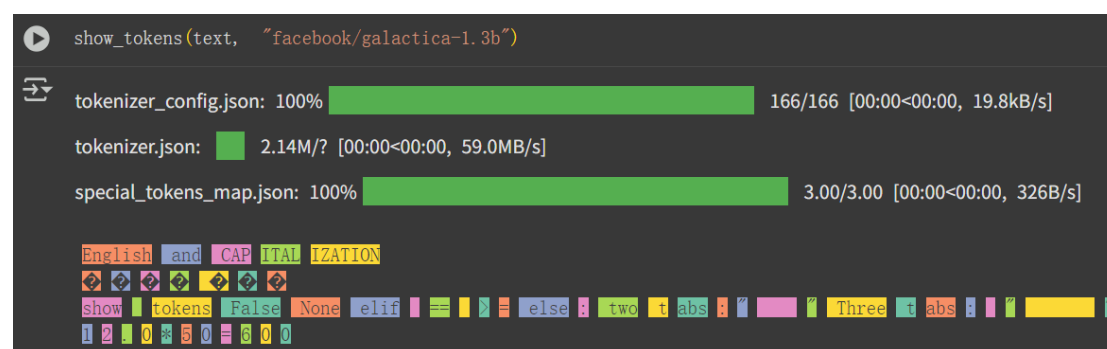
详见论文《Galactica: A large language model for science》。该模型专注于科学知识，并在许多科学论文和知识库上进行训练。它包含很多针对学术领域的特殊分词，比如 citation, reasoning, mathematics, amino acid sequences 氨基酸序列和 DNA 序列。

分词方法：BPE； 词表大小：50000；

特殊分词：<s>, <pad>, </s>, <unk>； citation 包含在两个特殊分词中间：

[START\_REF], [END\_REF]； 用于推理的特殊分词<work>。

Galactica 分词器的行为与 StarCoder2 类似，因为它也是为代码设计的。将空格序列用单个 token 表示。但是，它对制表符 tab 也采取了同样的处理方式。因此，它将两个制表符表示为单个 token。



```
show_tokens(text, "facebook/galactica-1.3b")
```

tokenizer\_config.json: 100% 166/166 [00:00<00:00, 19.8kB/s]

tokenizer.json: 2.14M/? [00:00<00:00, 59.0MB/s]

special\_tokens\_map.json: 100% 3.00/3.00 [00:00<00:00, 326B/s]

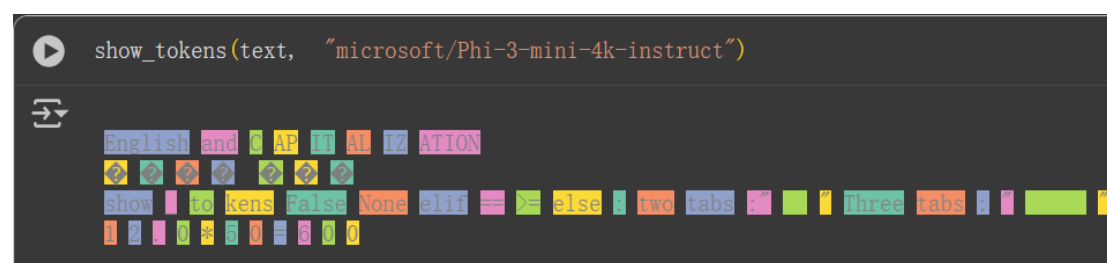
English and CAP ITAL IZATION

show tokens False None elif == >= else : two tabs : "Three tabs : "

## ⑨ Phi-3 (and Llama2)

Phi-3

分词方法：BPR； 词表大小：32000； 特殊分词：<|endoftext|>，聊天 tokens:<|user|><|assistant|><|system|>。由于聊天 LLM 在近几年流行，分词器为了适应这一趋势增加了对话中的轮次和每个 speaker 的身份 token。

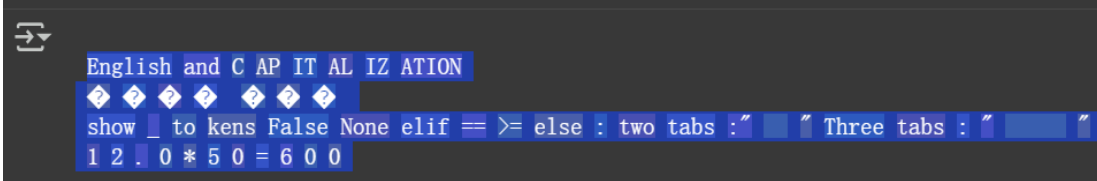


```
show_tokens(text, "microsoft/Phi-3-mini-4k-instruct")
```

English and AP IT AT IZ ATION

show tokens False None elif == >= else : two tabs : "Three tabs : "

```
show_tokens(text, "microsoft/Phi-3-mini-4k-instruct")
```



## 6. Tokenizer Properties 分词器属性

在自然语言处理和机器学习领域，Tokenizer Properties 指的是分词器（Tokenizers）所具有的特定配置、参数、能力和特征。这些属性决定了分词器如何处理文本以及生成什么样的输出。

上文介绍了九种不同的分词器，它们的分词策略各有特点，是什么决定了分词策略呢？和第三节中提到的相同，主要有三个方面：分词方法、初始化参数和用于训练分词器的数据来源。

### ① Tokenization methods 分词方法

目前存在多种分词方法，其中字节对编码（BPE）是较流行的一种。每种方法都阐述了一种算法，用于选择能最佳表示数据集的标记集合。我们可以在 Hugging Face 总结分词器的页面上找到所有这些方法的精彩概述。

[https://huggingface.co/docs/transformers/tokenizer\\_summary](https://huggingface.co/docs/transformers/tokenizer_summary)

### ② Tokenizer parameters 分词器参数

参数包括 Vocabulary size 词表大小、Special tokens 特殊分词、Capitalization 如何处理大写。

词表大小是指分词器的词汇表中应该保留多少 tokens，通常大小为 30K、50K，越来越多的词表大小达到了 100K。

常见的特殊分词大概有六类：Beginning of text token, End of text token, Padding token, Unknown token, CLS token, Masking token. 除此之外，设计者还可以添加其他的特殊分词来更好地解决所关注领域的问题，就像针对代码生成的模型 Galctica。

### ③ The domain of the data 数据的所属领域

即使选择相同的分词方法和参数，用来训练的数据不同，分词器的行为也会不同。例如专注于代码的模型通常会将关键词和缩进分别当做单个 token。

```
def add_numbers(a, b):
    """Add the two numbers `a` and `b`."""
    return a + b

def add_numbers(a, b):
    """Add the two numbers `a` and `b`."""
    return a + b
```

更多资料教程: [Introduction - Hugging Face LLM Course](#)、[Natural Language Processing with Transformers, Revised Edition \[Book\]](#)

## 二. Token 的嵌入分量

当我们在一个足够大的模型上训练模型时，它就会开始捕捉训练数据集中出现的复杂模式：如果训练数据包含大量英文文本，这种模式就会显现为一个能够表征和生成英语语言的模型。如果训练数据包含事实性信息（例如维基百科），该模型将具备生成一些事实性信息的能力（能够回答常识性问题）。

然后为这些 Tokens 找到一种最佳的数值表示形式，模型可以利用这些形式进行计算并恰当地建模文本中的模式，这些模式表现为模型在特定语言中的连贯性、编写代码的能力或其他能力。这就是嵌入 embedding 的作用，它们用来捕捉语言中的含义，是模式的数值表示空间。

### 1. 语言模型为其分词器的词汇表持有嵌入向量

分词器训练完成后将被用于其对应的语言模型的训练过程。如图 2-7 所示，语言模型为分词器词表中的每个 Token 持有一个嵌入向量。当我们下载预训练语言模型时，模型的一部分正是存储所有这些向量的嵌入矩阵。在训练过程开始前，这些向量与模型其他权重一样处于随机初始化状态，但通过训练过程，它们会被赋予实现特定功能所需的有效数值。

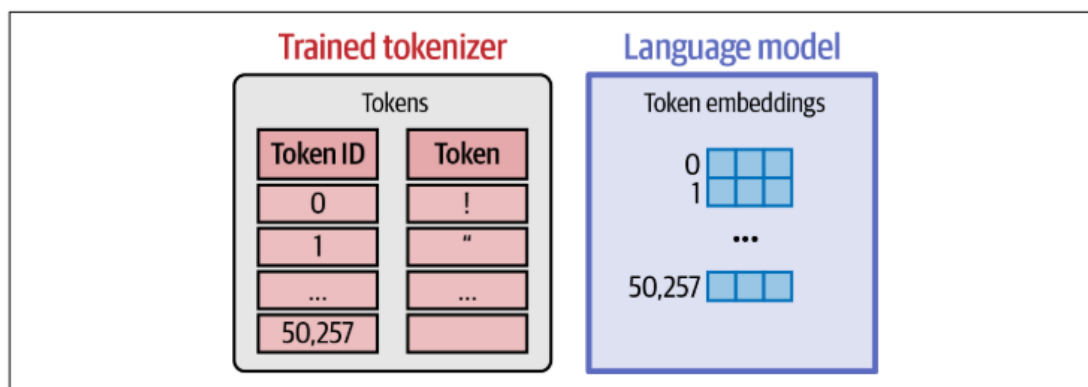


Figure 2-7. A language model holds an embedding vector associated with each token in its tokenizer.

## 2. 使用大语言模型创建具有上下文语境的单词嵌入

在了解了作为语言模型输入的 token 嵌入之后，我们接下来探讨语言模型如何生成更优质的词符嵌入。这是将语言模型用于文本表征的主要方式之一，它为命名实体识别或抽取式文本摘要（通过突出长文本中最关键的部分进行摘要，而非生成新文本作为摘要）等应用提供了技术支撑。

与使用静态向量表示每个 token 或单词不同，语言模型会生成上下文情境化词嵌入，如图 2-8 所示，即根据上下文语境用不同的向量表征同一个词语，例如在不同语境下，“bank” 分别表示“银行”和“河岸”时，经过 LLM 处理后生成的具有上下文语境的嵌入向量是不同的。

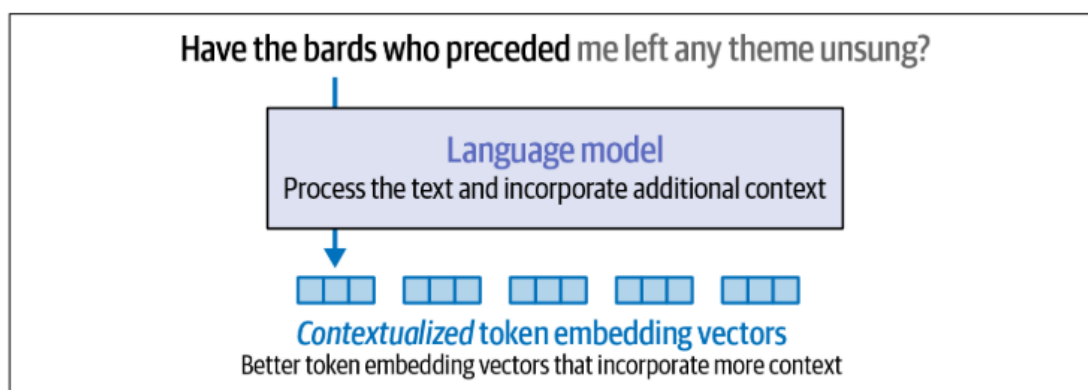


Figure 2-8. Language models produce contextualized token embeddings that improve on raw, static token embeddings.

如何生成上下文嵌入：



```

from transformers import AutoModel, AutoTokenizer

# 下载分词器
tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-base")

# 下载语言模型
model = AutoModel.from_pretrained("microsoft/deberta-v3-xsmall")

# 分词化，为句子生成token
tokens = tokenizer('Hello world', return_tensors='pt')

# 将处理后的 tokens 输入模型并获取输出
output = model(**tokens)[0]
# 索引[0]用于获取这个元组中的第一个也是最重要的元素——最后一层隐藏状态。

```

这个 output 张量就是句子“Hello world”的上下文词嵌入。每个 token 对应的向量都包含了其本身的信息以及它所在上下文的全部信息。

```

output.shape
# 这个输出的形状是(batch_size, sequence_length, hidden_size)
# 对于本例，batch_size 是1（一个句子），sequence_length是输入token
# 的数量（4，包括特殊token），hidden_size是模型的隐藏层维度（384，取决于模型配置）。

```

```
torch.Size([1, 4, 384])
```

每个 Token embedding 是一个有 384 个值的向量，也就是 384 维。

```

# 查看这4个Token分别是什么
for token in tokens['input_ids'][0]:
    print(tokenizer.decode(token))

```

```

[CLS]
Hello
world
[SEP]

```

[CLS]和[SEP]添加到字符串的开头和结尾。

查看完整 output，这是 LLM 的原始输出：

```

output
tensor([[[[-3.4816,  0.0861, -0.1819, ..., -0.0612, -0.3911,  0.3017],
          [ 0.1898,  0.3208, -0.2315, ...,  0.3714,  0.2478,  0.8048],
          [ 0.2071,  0.5036, -0.0485, ...,  1.2175, -0.2292,  0.8582],
          [-3.4278,  0.0645, -0.1427, ...,  0.0658, -0.4367,  0.3834]]],
        grad_fn=<NativeLayerNormBackward0>)]

```

总结上文中语言模型处理从原始句子变成富含上下文信息的向量表示的完整过程，我们可以得到下面这张图：

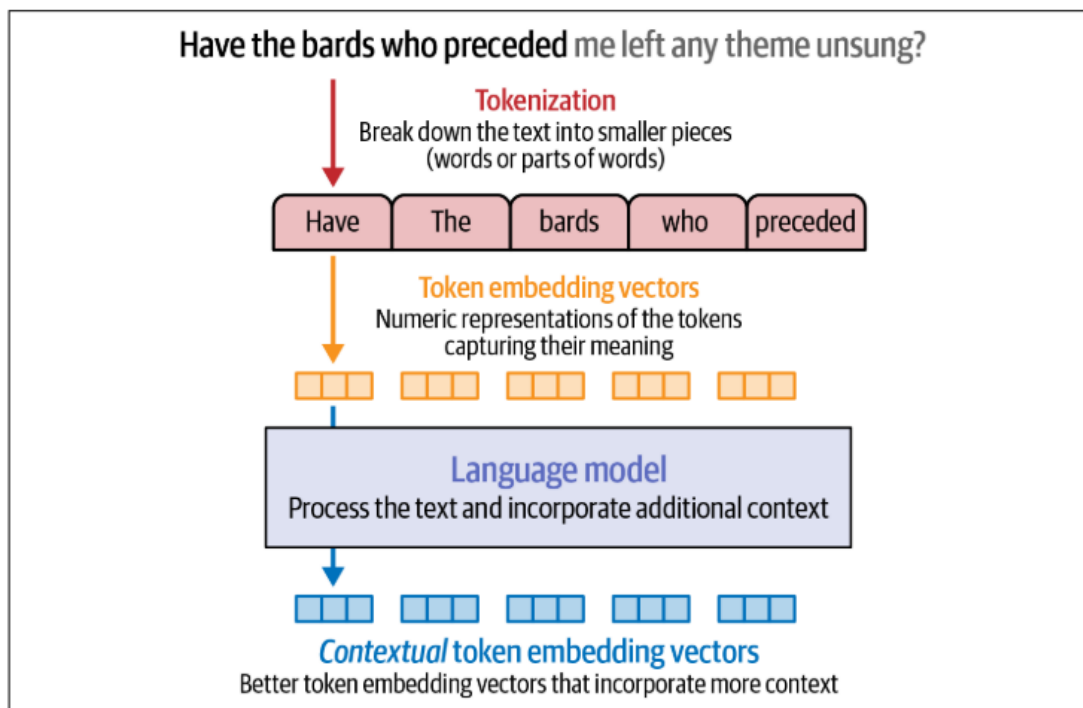


Figure 2-9. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

过程的核心思想是：语言模型能够根据一个词周围的词来动态地调整表示该词的向量，从而使这个词的表示更加精准。输入原始文本后，分词器将文本分解成多个 tokens，并将每个 token 映射成一个固定的嵌入向量，嵌入向量包含了这个词的通用语义信息，但它是静态的、没有上下文信息的。随后这些嵌入向量输入到 LLM 中，LLM 利用自注意力（Self-Attention）等机制，让序列中的每一个 token 都去“关注”序列中的所有其他 token（包括它自己）。通过这种机制，每个 token 的向量都融入了整个句子的上下文信息。最后，语言模型输出一套新的、富含上下文信息的向量序列。这些向量与输入的静态嵌入一一对应，但内容已经完全不同。

### 三. 文本嵌入（针对句子与完整文档）

许多 LLM 应用需要处理完整句子、段落甚至整个文本文档。这一需求催生了专门的文本嵌入模型——这类模型能够生成代表长文本片段的单一向量。我们可以将文本嵌入模型理解为：接收一段文本后，最终生成一个单一向量来表示该文本，并以某种有效形式捕捉其核心语义。图 2-10 展示了这一过程。

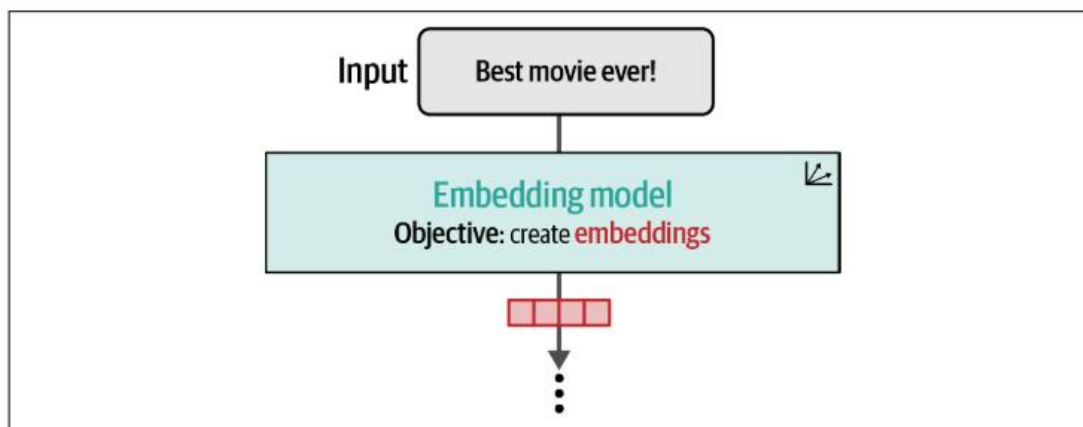


Figure 2-10. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

有多种产生文本嵌入向量的方式。最常见的方法之一是对模型生成的所有嵌入向量的值进行平均。然而，高质量的文本嵌入模型通常需要经过针对文本嵌入任务的专门训练。

我们可以使用 `sentence-transformers` 这个流行工具包来生成文本嵌入，该工具包能够调用预训练的嵌入模型。与前一章介绍的 `transformers` 库类似，这个工具包可用于加载公开可用的模型。

```
from sentence_transformers import SentenceTransformer

# Load model
model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')

# Convert text to text embeddings
vector = model.encode("Best movie ever!")
```

嵌入向量的维度取决于底层嵌入模型。打印下载的模型的向量维度：

```
vector.shape
(768,)
```

这个句子现在被编码在这个具有 768 个数值的维度的向量中。

#### 四. 超越大语言模型的词嵌入

嵌入技术的应用远不止于文本和语言生成领域，还被证明在推荐引擎和机器人技术等多个领域都具有重要价值。本节我们将探讨如何使用预训练的 `word2vec` 嵌入模型，并简要介绍其生成词嵌入的原理。

## 1. 使用预训练词嵌入

Gensim 是一个专注于“无监督主题建模”和“词向量表示”的强大工具包。我们可以用 Gensim 库下载预先训练好的单词嵌入模型，例如 word2vec 或 Glove。

注意，需要重新安装 numpy。

```
!pip uninstall gensim numpy scipy
!pip install gensim numpy scipy
```

下载在维基百科上训练的单词嵌入。

```
import gensim.downloader as api

# Download embeddings (66MB, glove, trained on wikipedia, vector size: 50)
# Other options include "word2vec-google-news-300"
# More options at https://github.com/RaRe-Technologies/gensim-data
model = api.load("glove-wiki-gigaword-50")

[=====] 100.0% 66.0/66.0MB downloaded
```

我们可以查看距离单词“king”最近的几个单词来探索嵌入空间。

```
model.most_similar([model['king']], topn=11)

[('king', 1.0000001192092896),
 ('prince', 0.8236179351806641),
 ('queen', 0.7839043140411377),
 ('ii', 0.7746230363845825),
 ('emperor', 0.7736247777938843),
 ('son', 0.766719400882721),
 ('uncle', 0.7627150416374207),
 ('kingdom', 0.7542161345481873),
 ('throne', 0.7539914846420288),
 ('brother', 0.7492411136627197),
 ('ruler', 0.7434253692626953)]
```

## 2. Word2vec 算法与 Contrastive Training 对比训练

论文《Efficient estimation of word representations in vector space》中描述的 word2vec 算法在 <https://jalamar.github.io/illustrated-word2vec/> 中有详细阐述。

与大型语言模型（LLM）类似，word2vec 同样基于文本生成的样本进行训练。Word2vec 的核心思想是：以弗兰克·赫伯特《沙丘》小说中的文本“Thou shalt not make a machine in the likeness of a human mind”为例，该算法采用滑

动窗口机制生成训练样本。若设置窗口大小为 2，即考虑中心词两侧各两个相邻词汇。中心词分别于左右两侧的两个单词组成四组单词对。随后在训练时，模型预测某一对词语是否经常出现在相同语境中（此处“语境”指训练数据集中多个句子的上下文环境）。若它们倾向于出现在相同语境则输出 1，否则输出 0。

如图 2-11 所示，中心词是 “not”，滑动窗口用于为 word2vec 算法生成训练样本，共生成四对样本 (not, thou), (not, shalt), (not, make), (not, a)。

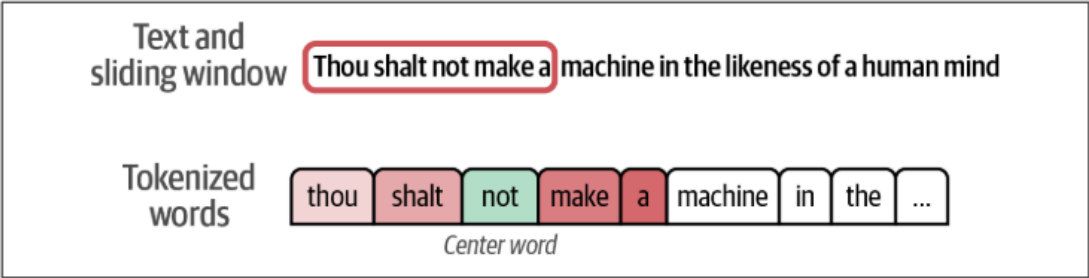


Figure 2-11. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

我们期望最终训练的模型能够对这种邻近关系进行分类，如果它接收的两个输入单词确实是邻居，则输出 1。这些训练示例如图 2-12 所示。

Training examples	Word1	Word2	Target
	Not	thou	1
	Not	shalt	1
	Not	make	1
	Not	a	1

Figure 2-12. Each generated training example shows a pair of neighboring words.

然而，如果我们有一个目标值全为 1 的数据集，那么模型可以通过一直输出 1 来应付训练。为了解决这个问题，我们需要向训练数据集中加入一些反例：通常不是相邻的单词对，如图 2-13 所示。

Word 1	Word 2	Target	Positive examples
not	thou	1	
not	shalt	1	
not	make	1	
not	a	1	Negative examples
thou	apothecary	0	
not	sublime	0	
make	def	0	
a	playback	0	

Figure 2-13. We need to present our models with negative examples: words that are not usually neighbors. A better model is able to better distinguish between the positive and negative examples.

事实上，在选择负样本时我们无需过于讲究方法论。许多实用模型仅仅通过从随机生成的样本中识别正样本的能力就能训练成功。因此在本例中，我们只需随机选取词汇加入数据集，并标记它们不属于相邻词汇（当模型遇到这些样本时应输出 0）。

在开始基于该数据集训练神经网络之前，我们需要做出几个分词决策：正如我们在 LLM 分词器中看到的，这些决策包括：如何处理大小写和标点符号，以及词表中需要包含多少 token。接着我们为每个 token 创建一个嵌入向量并进行随机初始化（如图 2-15 所示）。实际应用中，这些向量会组成一个维度为 **词表大小 × 嵌入维度** 的矩阵。

Token	Token embedding
thou	<div><div></div><div></div><div></div></div>
shalt	<div><div></div><div></div><div></div></div>
make	<div><div></div><div></div><div></div></div>
a	<div><div></div><div></div><div></div></div>
not	<div><div></div><div></div><div></div></div>
apothecary	<div><div></div><div></div><div></div></div>
sublime	<div><div></div><div></div><div></div></div>
def	<div><div></div><div></div><div></div></div>
playback	<div><div></div><div></div><div></div></div>

Figure 2-15. A vocabulary of words and their starting, random, uninitialized embedding vectors.

训练时，模型会获取两个嵌入向量并预测它们是否相关。基于预测结果，模型会不断更新参数，确保模型输出的预测概率更符合标签。训练结束后，词表中的所有 token 都有自己独特的嵌入向量，每个嵌入向量的数值都不同。我们可以这样理解：对于意义相近的单词，它们在向量空间中的“距离”更近，就像上一节代

码运行结果中的“king”和“prince”很接近一样。

这种接收两个向量并预测它们是否具有特定关系的模型架构，堪称机器学习领域最强大的思想之一，其在语言模型中的有效性已被反复验证。我们将在第十章专门探讨这一概念。该思想同样是连接文本与图像等多模态领域的关键桥梁。

## 五. 推荐系统 Recommendation Systems 中的嵌入表示

### 1. 通过嵌入的方式推荐歌曲

正如我们已经提到的，嵌入的概念在许多其他领域中都很有用。例如，在工业中，它被广泛用于 Recommendation Systems 推荐系统。

在本节中，我们将运用 word2vec 算法，基于人工创建的音乐歌单为歌曲生成嵌入向量。设想将每首歌曲视作一个词语或 token，将每个歌单视为一个句子。通过这些嵌入向量，我们可以推荐那些经常在歌单中共同出现的相似歌曲，就像预测两个单词是否相邻。

使用的数据集是由康奈尔大学 Shuo Chen 收集的，该数据集包含来自美国数百家广播电台的歌单记录。图 2-17 直观展示了该数据集的结构特征。



Figure 2-17. For song embeddings that capture song similarity we'll use a dataset made up of a collection of playlists, each containing a list of songs.

### 2. 训练一个歌曲嵌入模型

这段代码完成后，我们得到了两个关键变量：

**playlists:** 一个包含了成千上万个播放列表的列表，每个播放列表是几首歌曲 ID 的序列。这就是用来训练“歌曲版 word2vec”模型的“语料库”。模型将学习这些 ID 的共现模式（即哪些“歌曲”经常在同一个“播放列表”中出现）。

**songs\_df:** 一个 Pandas DataFrame，其索引是歌曲 ID，列是歌曲标题和艺术家。这是一个“字典”或“查找表”。当模型为我们推荐出一系列相似的歌曲 ID 后，我们可以用它来将这些 ID 转换回人类可读的歌曲名称和艺术家。







我们可以查看 2172 是什么歌曲：因此推荐的歌曲应该与重金属摇滚相关。

```
print(songs_df.iloc[2172])
```

```
title    Fade To Black
artist    Metallica
Name: 2172, dtype: object
```

```
import numpy as np
```

```
# 根据输入的歌曲ID，使用训练好的word2vec模型（model）来查找最相似
# 的5首歌曲，并返回这些相似歌曲的详细信息（歌名和艺术家）
def print_recommendations(song_id):
    similar_songs = np.array(
        model.wv.most_similar(positive=str(song_id), topn=5) # 查找最相似的五首歌曲
    )[:, 0] # 提取出所有相似歌曲的ID，丢弃相似度得分，
           # 结果是一个包含五首歌曲ID的一维数组

    return songs_df.iloc[similar_songs] # 返回结果是一个DataFrame，
                                       # 包含5首最相似歌曲的ID、标题和艺术家信息

# Extract recommendations
print_recommendations(2172)
```

```
id          title          artist
2849  Run To The Hills  Iron Maiden
3167    Unchained      Van Halen
3094  Breaking The Law  Judas Priest
11473  Little Guitars   Van Halen
2704  Over The Mountain Ozzy Osbourne
```

再尝试打印与歌曲 842 最相似的五首歌：

```
print_recommendations(842)
```

```
id          title          artist
1560              In Da Club      50 Cent
27081  Give Me Everything (w\ Ne-Yo, Afrojack & Nayer)  Pitbull
413    If I Ruled The World (Imagine That) (w\ Laury...    Nas
211              Hypnotize  The Notorious B.I.G.
330              Hate It Or Love It (w\ 50 Cent)      The Game
```