02.《自然语言处理实战·第2版》第2章 思想的标记：自然语言词汇

本章内容

- 将文本解析为词和n元语法（标记）
- 对标点符号、表情符号甚至中文字符进行标记化
- 通过词干提取、词形还原和大小写折叠来整合词汇表
- 构建自然语言文本的结构化数值表示
- 对文本的情感和亲社会意图进行评分
- 使用字符频率分析优化你的标记词汇表
- 处理可变长度的词和标记序列

所以你想利用自然语言处理（NLP）的力量来帮助拯救世界？无论你希望自然语言处理流水线执行什么任务，它都需要对文本进行计算。为此，你需要一种方法来用数值数据结构表示文本。自然语言处理流水线中将文本分解为更小单位并可用于数值表示的部分称为标记分析器（tokenizer）。标记分析器将非结构化数据、自然语言文本分解为信息块，这些信息块可以作为离散元素进行计数。这些文档中标记出现次数的计数可以直接用作表示该文档的向量。这立即将非结构化字符串（文本文档）转换为适合机器学习的数值数据结构。

标记化（tokenization）最基本的用途是它可以用来计算文档的统计表示。关于标记的统计数据通常是你进行关键词检测、全文搜索和信息检索所需的全部内容，我们将在第3、5、10和12章中讨论这些内容。你甚至可以使用文本搜索来构建客户支持聊天机器人，在你的文档或常见问题（FAQ）列表中查找客户问题的答案。聊天机器人在知道去哪里寻找答案之前无法回答你的问题。搜索是许多最先进应用的基础，如对话式人工智能和开放域问答。这种统计表示还可以用于分类和比较不同的文本片段，比如检测文本情感（见第3章和第4章）。

但这并不是你可以用标记做的唯一事情。你可以提出一种数值表示来帮助反映标记的含义，例如第6章中讨论的词向量。通过结合个别标记的含义，你可以创建一个计算机知道如何处理的文本表示。

标记分析器构成了几乎所有自然语言处理流水线的基础。即使是最大的大型语言模型（LLMs），如ChatGPT背后的模型，也会将文本分解为标记，并通过预测序列中的下一个标记来工作。由于将文本分解为标记是自然语言处理流水线的第一步，它可能对流水线的其余部分产生重大影响。在本章中，你将学习可用于将文本转换为标记序列的不同技术。

2.1 标记和标记化

在自然语言处理中，标记化（tokenization）是一种特殊的文档分割方式。分割将文本拆分成更小的块或段落。文本的这些片段包含的信息少于整体。文档可以被分割成段落，段落分割成句子，句子分割成短语，短语分割成标记（通常是词和标点符号）。在本章中，我们专注于使用标记分析器将文本分割成标记。

标记可以是你想要作为思想和情感数据包处理的几乎任何文本块。特定语言的有效标记集合被称为该语言的词汇表，或者更正式地称为其词库（lexicon）。语言学和自然语言处理研究人员使用术语"词库"来指代一组自然语言标记的集合。"词汇表"（vocabulary）是指代一组自然语言词或标记集合的更自然方式，所以这是我们在这里要使用的术语。

在本章的第一部分，你的标记将是单词、标点符号，甚至是表意符号和标志符号，如汉字、表情符号（emoji）和情绪符号（emoticon）。然后我们将讨论现代自然语言处理中越来越多使用的其他类型的标记化，特别是用于操作大型语言模型（LLMs）。在本书后面部分，你将看到你可以使用这些相同的技术在任何离散序列中找到意义单元。例如，你的标记可以是由字节序列表示的ASCII字符，可能带有ASCII表情符号，或者它们可以是Unicode表情符号、数学符号、埃及象形文字，甚至是DNA或RNA序列中的单个氨基酸。自然语言标记序列无处不在……甚至在你体内。

2.1.1 标记分析器工具箱

现在你对存在哪些类型的标记有了更好的理解，是时候开始真正的标记化了。你可以从几种标记分析器实现中进行选择：

1. Python–str.split和re.split
2. 自然语言工具包（NLTK）–TreebankWordTokenizer和TweetTokenizer
3. spaCy–快速且生产环境就绪的标记化
4. Stanford CoreNLP–语言学准确性高，需要Java解释器
5. Hugging Face–BertTokenizer，一种WordPiece标记分析器

2.1.2 最简单的标记分析器

对句子进行标记化的最简单方法是使用字符串中的空白作为词的"分隔符"。在Python中，这可以通过标准库方法split来实现。

假设你的自然语言处理流水线需要解析来自WikiQuote.org的引文，并且在处理一个名为《偷书贼》（The Book Thief）的引文时遇到了麻烦。

代码清单2.1 来自《偷书贼》的示例引文分割成标记 Python

```
text = ("Trust me, though, the words were on their way, and when "
... "they arrived, Liesel would hold them in her hands like "
... "the clouds, and she would wring them out, like the rain.")
# str.split()是你快速而粗糙的标记分析器。
tokens = text.split()
tokens[:8]
['Trust', 'me,', 'though,', 'the', 'words', 'were', 'on', 'their']
```

正如你所看到的，这个内置Python方法在标记化这个句子上做得还可以。它唯一的"错误"是将逗号包含在标记中。这会阻止你的关键词检测器检测到一些重要的标记：['me', 'though', 'way', 'arrived', 'clouds', 'out', 'rain']。单词clouds和rain对于这段文本的含义很重要，所以你需要用你的标记分析器做得更好一些，以确保捕获所有重要的单词并"保留"它们（就像《偷书贼》中的Liesel一样）。

2.1.3 基于规则的标记化

事实证明，解决将标点符号从单词中分离出来的挑战有一个简单的方法。你可以使用正则表达式标记分析器来创建规则，以处理常见的标点符号模式。以下是一个可以用来处理标点符号"附着物"的正则表达式。同时，这个正则表达式会智能处理包含内部标点符号的单词，例如含有撇号的所有格词和缩写词。

你将使用正则表达式来对彼得·沃茨（Peter Watts）的书《盲视》（Blindsight）中的一些文本进行标记化。这段文本描述了最适应的人类如何在自然选择（和外星人入侵）中生存下来。对于你的标记分析器也是如此。你想找到一个能充分解决问题的标记分析器，而不是完美的标记分析器。你可能甚至无法猜测什么是正确的或最适合的标记。你需要一个准确性数字来评估你的自然语言处理流水线，那将告诉你哪个标记分析器应该在你的选择过程中生存下来。以下例子应该能帮助你开始培养对正则表达式标记分析器应用的直觉：

```
>>> import re
# 前瞻模式（lookahead pattern） `(?:'\w+)?` 用于检测一个词中
# 是否包含【一个单引号，后跟一个或多个字母】的结构。
>>> pattern = r'\w+(?:\'\w+)?|[^\w\s]'
>>> texts = [text]
>>> texts.append("There's no such thing as survival of the fittest. "
... "Survival of the most adequate, maybe.")
>>> tokens = list(re.findall(pattern, texts[-1]))
>>> tokens[:8]
["There's", 'no', 'such', 'thing', 'as', 'survival', 'of', 'the']
>>> tokens[8:16]
['fittest', '.', 'Survival', 'of', 'the', 'most', 'adequate', ',']
>>> tokens[16:]
['maybe', '.']
```

这样好多了。现在，标记分析器将标点符号与单词末尾分开，但它不会分解包含内部标点符号的单词，例如标记There's中的撇号。所以所有这些单词都按照你想要的方式进行了标记化：There's、fittest和maybe。这个正则表达式标记分析器在处理缩写时也能正常工作，即使撇号后面有多个字母，如ya'll、she'll和what've。这甚至对撇号错误也有效，如can"t、she,ll和what`ve。但如果你的文本包含罕见的双重缩写，如couldn't've、ya'll'll和y'ain't，这种对内部标点符号的宽松匹配可能并不是你想要的。

> 提示：你可以使用正则表达式r'\w+(?:'\w+){0,2}|[^\w\s]'来适应双重缩写。

要记住的主要思想是：**无论你多么精心地设计标记分析器，它可能都会破坏原始文本中的一定量信息**。当你切割文本时，你只希望确保留在"剪辑室地板上"的信息对于你的流水线做好工作是无关紧要的。此外，思考你的下游自然语言处理算法也很有帮助。之后，你可能会配置大小写折叠、词干提取、词形还原、同义词替换或向量化计数算法。当你这样做时，你必须思考你的标记分析器在做什么，以确保你的整个流水线在协同工作，实现你期望的输出。

看看这段简短文本中词汇表按字典顺序排序的前几个标记：

```
>>> import numpy as np

# 将列表强制转换（coerce）为集合（set），以确保词汇表中只包含唯一的标记（token），不含重复项。
>>> vocab = sorted(set(tokens))

# 进行字典序排序（lexicographical sort），使得标点符号排在字母之前，且大写字母排在小写字母之前。
>>> ' '.join(vocab[:12])
", . Survival There's adequate as fittest maybe most no of such"
>>> num_tokens = len(tokens)
>>> num_tokens
18
>>> vocab_size = len(vocab)
>>> vocab_size
15
```

你可以看到为什么你可能想要将所有标记小写化，这样Survival就会被识别为与survival相同的词。出于类似的原因，你可能希望有一个同义词替换算法，将There's替换为There is。然而，这只有在你的标记分析器保持缩写和所有格撇号附着在它们的父标记上时才有效。

> 提示：每当你的流水线似乎对特定文本工作不佳时，请务必查看你的词汇表。你可能需要修改标记分析器，以确保它能"看到"所有需要的标记，以便在你的自然语言处理任务中表现良好。

2.1.4 spaCy

也许你不希望你的正则表达式标记分析器保持缩写词的完整性。或许，你希望将isn't识别为两个独立的词：is和n't。这样，你就可以将同义词n't和not合并为单个标记。这使你的自然语言处理流水线能够理解，例如the ice cream isn't bad与the ice cream is not bad具有相同的含义。对于某些应用，如全文搜索、意图识别和情感分析，你需要能够解除缩写或扩展缩写词。通过分割缩写词，你可以使用同义词替换或缩写扩展来提高搜索引擎的召回率和情感分析的准确性。

注意：我们将在本章后面讨论大小写折叠、词干提取、词形还原和同义词替换。对于作者归属识别、风格迁移或文本指纹等应用，请谨慎使用这些技术。你希望你的作者归属识别或风格迁移流水线忠实于作者的写作风格和他们使用的单词的确切拼写。

spaCy将标记分析器直接集成到其最先进的自然语言理解（NLU）流水线中，并在应用规则分割标记时为标记添加几个额外的标签。因此，**spaCy通常是你需要使用的第一个也是最后一个标记分析**

器。

让我们看看spaCy如何处理我们收集的深度思想家引言：

```python
# 如果这是你第一次使用 spaCy，应通过 spacy.cli.download('en_core_web_sm') 下载小型语言模型。
>>> import spacy

# 为了避免重复下载语言模型，可以使用 from spacy_language_model import nlp 直接加载。
>>> spacy.cli.download('en_core_web_sm')

# 这里的 "sm" 表示 **小型（small）**模型（17 MB）
# "md" 表示 **中型（medium）**模型（45 MB），
# "lg" 表示 **大型（large）**模型（780 MB）。
>>> nlp = spacy.load('en_core_web_sm')
>>> doc = nlp(texts[-1])
>>> type(doc)
spacy.tokens.doc.Doc
>>> tokens = [tok.text for tok in doc]
>>> tokens[:9]
['There', "'s", 'no', 'such', 'thing', 'as', 'survival', 'of', 'the']
>>> tokens[9:17]
['fittest', '.', 'Survival', 'of', 'the', 'most', 'adequate', ',']
```

那种标记化在对比学术论文或与同事的结果做对比时可能更有用。spaCy在底层做了更多工作。你下载的那个小型语言模型还使用一些句子边界检测（sentence boundary detection）规则来识别句子分隔。语言模型是正则表达式和有限状态自动机或规则的集合，很像你在英语课上学到的语法和拼写规则。这些规则用于标记化和标记单词的算法中，为词标注有用的信息，如它们的词性（part of speech，POS）以及在表示词之间关系的句法树中的位置：

```python
>>> from spacy import displacy

# 第一句以 “There’s no such thing … .” 开头
>>> sentence = list(doc.sents)[0]

# 将参数 jupyter=False 设置为返回一个 SVG 字符串，你可以将其保存到磁盘上
>>> svg = displacy.render(sentence, style="dep",
... jupyter=False)

# 如果你使用的是 IPython 控制台，可以在本地硬盘上浏览这个 SVG 文件
>>> open('sentence_diagram.svg', 'w').write(svg)

# 你也可以通过 Web 服务器与句子图的 HTML 和 SVG 渲染结果进行交互。
>>> # displacy.serve(sentence, style="dep")
>>> # !firefox 127.0.0.1:5000

# 当设置为 jupyter=None 时，displaCy 会尽可能以内嵌方式显示 SVG
>>> displacy.render(sentence, style="dep")
```

从displaCy创建和查看句子图表有三种方式：在网页浏览器中显示动态HTML/SVG文件，在硬盘上保存静态SVG文件，或在Jupyter Notebook中内嵌HTML对象。如果你在本地硬盘上浏览sentence_diagram.svg文件或访问localhost:5000服务器，你应该能看到一个句子图表，这可能比你在学校里能制作的还要好。

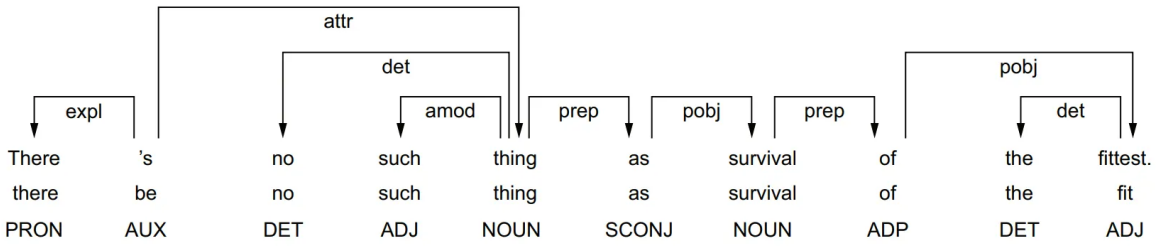

Figure 2.1 The sentence diagram produced after running spaCy

图2.1显示了关于生存那个句子中每个标记的词性和词形。displaCy图表还包括一个由连接词语的弧线表示的依存树（dependency tree），这样你就能看到句子中概念的逻辑嵌套和分支。你能找到依存树的根（root）吗--即不依赖其他词来修饰其含义的标记？寻找那个没有箭头指向它且只有箭尾向依赖词弧出的单词。

你可以看到spaCy做的远不止简单地将文本分割成标记。它还识别句子边界，自动将文本分段为句子，并用各种属性标记标记，如它们的词性，甚至是它们在句子语法中的角色。你可以看到displaCy在每个标记的文字下方显示的词形。在本章后面，我们将解释词形还原

（lemmatization）、大小写折叠（case folding）和其他词汇压缩方法如何对某些应用有所帮助。

所以，就准确性和一些"内置"功能（如所有那些用于词形和依存关系的标记标签）而言，spaCy似乎非常优秀。但速度如何呢?

2.1.5 寻找最快的标记分析器

spaCy能在大约5秒内解析本书一章的AsciiDoc文本。首先，下载本章的AsciiDoc文本文件：

```python
>>> import requests
>>> text = requests.get('https://proai.org/nlpia2-ch2.adoc').text

# 先将数值除以 10,000 再进行四舍五入，这样在文本内容被修改时，doctest（文档测试）仍能通过
>>> f'{round(len(text) / 10_000)}0k'
'60k'
```

我们写下这句话的AsciiDoc文件中大约有160,000个ASCII字符。这在词/秒（标准标记分析器速度基准）方面意味着什么呢?

```python
>>> import spacy
>>> nlp = spacy.load('en_core_web_sm')

# %timeit 是一个 魔法函数（magic function），
# 可在 Jupyter Notebook、Jupyter Console 和 IPython 中使用
>>> %timeit nlp(text)
4.67 s ± 45.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> f'{round(len(text) / 10_000)}0k'
'160k'
>>> doc = nlp(text)
>>> f'{round(len(list(doc)) / 10_000)}0k'
'30k'

# kWPS 表示每秒处理的千词数量（千词标记数，thousands of words/tokens per second）
>>> f'{round(len(doc) / 1_000 / 4.67)}kWPS'
'7kWPS'
```

这差不多是5秒处理约150,000个字符或34,000个英语和Python文本单词，也就是每秒约7,000个单词。

对于个人项目来说，这可能看起来足够快，但在一个医疗记录总结项目中，我们需要处理数千个大型文档，其文本量与本书整体相当。而且在我们的医疗记录总结流水线中，延迟是项目的关键指标。因此，这个全功能spaCy流水线至少需要五天时间来处理10,000本书，例如NLPiA或10,000名患者的典型医疗记录。如果这对你的应用来说不够快，你可以禁用任何你不需要的spaCy流水线的标记功能：

```python
# pipe_names 属性列出了当前 spaCy NLP 流水线中启用的所有处理组件
>>> nlp.pipe_names
['tok2vec', 'tagger', 'parser', 'attribute_ruler', 'lemmatizer', 'ner']
>>> nlp = spacy.load('en_core_web_sm', disable=nlp.pipe_names)
>>> %timeit nlp(text)
199 ms ± 6.63 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

你可以禁用不需要的流水线元素来加速标记分析器：

- tok2vec—— 词嵌入（Word embeddings）
- tagger——词性（.pos和.pos_）
- parser——句法树角色
- attribute_ruler——细粒度词性和其他标签
- lemmatizer——词形还原标记器
- ner——命名实体识别标记器

NLTK的word_tokenize方法通常被用作标记分析器基准速度比较中的标杆：

```python
>>> import nltk
>>> nltk.download('punkt')
True
>>> from nltk.tokenize import word_tokenize
>>> %timeit word_tokenize(text)
156 ms ± 1.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> tokens = word_tokenize(text)
>>> f'{round(len(tokens) / 10_000)}0k'
'10k'
```

你可能认为已经找到了标记分析器竞赛的冠军？别急。你的正则表达式标记分析器规则相当简单，所以它也应该运行得相当快：

```python
>>> pattern = r'\w+(?:\'\w+)?|[^\w\s]'
# 尝试使用 re.compile 进行预编译（precompiling），
# 你会从中了解 Python 核心开发者的设计有多么巧妙
>>> tokens = re.findall(pattern, text)
>>> f'{round(len(tokens) / 10_000)}0k'
'20k'
>>> %timeit re.findall(pattern, text)
8.77 ms ± 29.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

这并不令人惊讶。正则表达式可以在Python中的低级C例程中被编译并高效运行。

> 提示：当速度比准确性更重要时，使用正则表达式标记分析器。如果你不需要spaCy和其他流水线提供的额外语言学标签，你的标记分析器不需要浪费时间去计算这些标签。每次你使用re或regex包中的正则表达式时，其编译和优化版本会被缓存在RAM中。所以通常不需要预编译（使用re.compile()）你的正则表达式。

2.2 超越标记

将词视为不可分割的意义和思想的原子块可能感觉很自然。然而，你确实发现有些词并不能明确地在空格或标点符号处分割。而且有许多你希望保持在一起的复合词或命名实体内部往往包含了空格。

词并不是我们可以用作标记的唯一意义单元。稍微想一想，对你来说，一个词或标记代表什么。它是代表一个单一概念还是一些模糊的概念云？你能总是识别出一个词在哪里开始和结束吗？自然语言中的词像编程语言关键词那样有精确的拼写、定义和使用语法规则吗？你能编写可靠识别词的软件吗？

你可能想将词划分为更小的、有意义的部分。词片段，如前缀pre、后缀fix和内部音节la，都有意义。你可以使用这些词片段将你学到的关于一个词的含义转移到词汇表中的另一个相似词。你的自然语言理解（NLU）流水线甚至可以使用这些片段来理解新词，而你的自然语言生成（NLG）流水线可以使用这些片段创建新词，简洁地捕捉集体意识中流传的想法或模因。

你的流水线可以将词分解成更小的片段。字母、字符和字形也传达情感和意义！我们尚未找到思想数据包的完美编码，况且机器的计算方式也与大脑不同。我们人类使用词和词组（复合词）相互解

释想法和语言，但机器常常能看到我们忽略的字符序列中的模式。而且为了让机器能够将庞大的词汇表压缩到有限的RAM中，对自然语言存在更高效的编码方式。

高效计算的最佳标记与我们人类使用的思想数据包（词）不同。字节对编码（Byte pair encoding，BPE）、词片段编码（Word piece encoding，WPE）和句子片段编码（Sentence piece encoding，SPE）都可以帮助机器更高效地使用自然语言，但这些对人类而言就不那么易于理解了。如果你想要一种可解释的编码，可以使用前几节中的标记分析器。如果你想要更灵活、更准确的文本预测和生成，那么对你来说BPE、WPE或SPE可能更好。

那么隐形或隐含的词呢？你能想到单词命令"Don't!"暗示的额外单词吗？如果你能强迫自己像机器一样思考，然后再切换回像人类一样思考，你可能会意识到这个命令中有三个隐形词："Don't!" "Don't you do that!"和"You, do not do that!" 这至少有三个隐藏的意义数据包，总共有五个你希望机器知道的标记。

但现在不要担心隐形词。在本章中，你只需要一个能够识别拼写出来的词的标记分析器。你将在第4章及以后关注隐含词、内涵甚至是意义本身。

你的自然语言处理流水线可以从这四个选项之一作为你的标记开始：

- 字符（ Characters ）——ASCII字符或多字节Unicode字符
- 子词（ Subwords ）——音节和常见字符簇（即词片段）
- 词（ Words ）——词典词或其词根（即词干或词形）
- 句子片段（ Sentence pieces ）——短的、常见的词和多词片段

随着你在这个列表中向下移动，你的词汇量会增加，你的自然语言处理流水线将需要越来越多的数据来训练。基于字符的自然语言处理流水线，通常用于需要从适量示例中泛化的翻译问题、或自然语言生成任务。基于字符的自然语言处理流水线通常需要少于200个可能的标记来处理许多拉丁语系的语言。这种小型词汇表确保基于字节和字符的自然语言处理流水线可以处理新的未见过的测试示例，而不会出现太多无意义的词汇外（Out-of-Vocabulary，OOV）标记。

对于基于词的自然语言处理流水线，在决定是否"计数"标记之前，你的流水线需要开始注意标记的使用频率。但即使你确保你的流水线关注经常出现的词，你最终可能得到一个与典型词典一样大的词汇表——2万到5万个词。

子词(*Subwords* or Word pieces)是大多数深度学习自然语言处理流水线的最佳标记选择。子词标记分析器内置于许多最先进的Transformer流水线中。词(*Words*)是任何语言学项目或学术研究的首选标记，这些研究的结果需要可解释和可说明。

句子片段(*Sentence pieces*)将子词算法推向极致。句子片段标记分析器允许你的算法将多个词片段组合成单个标记，有时可以跨越多个词。句子片段的唯一硬性限制是它们不会超过句子的结尾。

这确保了标记的含义只与单一连贯思想相关联，并且在单句以及更长文档上都有用。由于本书将大量涉及深度学习模型，让我们深入研究子词或WordPiece标记化的机制。

2.2.1 WordPiece标记分析器

想一想，我们能否通过相邻字符构建词，而不是一遇到空格或标点符号就将文本拆分。与其将字符串切分为标记（tokens），不如让分词器（tokenizer）去识别经常相邻出现的字符组合，比如 `i` 通常出现在 `e` 前面。你可以将属于同一结构的字符或字符序列配对组合，这些字符块就可以作为你的标记。NLP 流水线只关心标记的统计特征，而我们希望这些统计特征能够贴合我们对“词”这一语言单位的认知。

这些字符序列中，很多将是完整的单词，甚至是复合词；但也有很多只是词的一部分。事实上，所有子词分词器（subword tokenizer）都会在词表中保留每个单独字符的标记，这样一来，只要新文本中没有出现新的字符，就永远不会需要使用 OOV（out-of-vocabulary，词表外）标记。子词分词器的目标是最优地将字符聚合为标记。通过统计字符 n-gram（n元组）的频率，这些算法可以识别出合适的子词，甚至句子片段，作为理想的标记。

用字符聚类来识别单词在直觉上可能有些奇怪。但对于机器而言，文本中唯一一致、明确的结构边界是字节或字符之间的分隔。此外，字符的共现频率可以帮助机器从中识别出带有语义信息的子词标记，比如音节或复合词的组成部分。

在英语中，即便是单个字母，也常常带有微妙的情感（情绪倾向）和语义暗示。然而，英语中只有 26 个独特的字母，这使得单个字母很难专门表达某一特定话题或情绪。尽管如此，精明的市场营销人员知道，有些字母比其他字母“更酷”。品牌常常会选择不常见的字母（比如 Q、Y、Z）来命名，以营造出现代或科技感的形象（如 Lyft 或 Cheez-Its）。这也有助于搜索引擎优化（SEO），因为这些罕见字母更容易在海量公司或产品名中脱颖而出。你的 NLP 流水线会捕捉到这些语义、感情色彩与意图的线索，而标记统计工具将为机器提供推理这些字符组合含义所需的数据支持。

使用子词分词器的唯一缺点是：它们需要多次遍历语料库，以便最终收敛到一个最优的词表和分词器。这就像 `CountVectorizer` 一样，子词分词器也必须在你的语料上进行训练或拟合。实际上，在接下来的部分中，你将使用 `CountVectorizer` 来了解子词分词器是如何工作的。

子词分词有两种主流方法：BPE 与 WordPiece

字节对编码（BPE，Byte-Pair Encoding）

在本书的上一版中，我们坚持认为**词**是英语中最小的语义单位。但随着 Transformer 及其他使用 BPE 技术的深度学习模型的兴起，我们改变了看法。

基于字符的子词分词器已被证明在大多数 NLP 问题上更加灵活、鲁棒。通过从 Unicode 多字节字符出发构建词表，你可以创建一个能够处理所有自然语言字符串的词表，规模甚至可以少至 50,000 个标记。

你可能认为 Unicode 字符是自然语言中最小的语义单元。对人类来说也许是这样，但对机器来说则不然。正如 BPE 名称所暗示的那样，字符甚至不一定是词表中最小的单位。你可以将字符再分解为 8 位字节（byte）。GPT-2 就使用了**基于字节的 BPE 分词器（byte-level BPE tokenizer）**，它通过字节组合自然构建出所需的 Unicode 字符。虽然为了正确处理 Unicode 标点符号还需一些特殊规则，但对字符级 BPE 算法本身并不需要其他调整。

基于字节的 BPE 分词器允许你用最小仅 256 个标记的基础词表表示所有可能的文本。GPT-2 就使用了约 50,000 个多字节合并标记和 256 个单独字节标记，便可实现先进的语言理解效果。GPT-4 模型据传也采用了类似的词表规模：大约 50,000 个标记。

你可以将 BPE 分词算法想象成社交网络中的红娘：它将那些经常紧邻出现的字符配对，并为它们创建新的复合标记。BPE 还可以将多个字符标记再配对成更长的标记，只要这些组合在文本中也足够常见。它会一直进行这种合并，直到你设定的词表容量上限被用尽。

BPE 正在改变我们对自然语言中“标记”这一概念的认知。NLP 工程师正在**让数据说话**。在构建 NLP 流水线时，统计思维往往比人类直觉更有效。机器能看出多数人是如何使用语言的，而人类往往只能意识到自己在使用特定词语或音节时的主观意图。Transformer 模型在某些语言理解与生成任务中，已经超越人类读者与写作者——包括从子词标记中挖掘语义的能力。

你还没有处理的一个问题是：当 NLP 流水线首次遇到一个新词时该怎么办？在前面的例子中，我们只是不断将新词加入词表。但在真实应用中，你的模型通常会基于一个初始语料库训练，这些语料可能并不覆盖你之后会遇到的所有词。如果某个词不在初始词表中，那么你就没有办法为它统计词频。因此，在训练初始分词器时，通常会为 **OOV（词表外）标记保留一个专门的槽位（slot，也可以理解为维度）**。比如，如果你的语料库中没有出现过“Aphra”这个名字，那么这个词的所有统计都会被归入 OOV 维度，像“Amandine”这样的罕见词也是如此。

为了解决这个问题并公平对待“Aphra”，你可以使用 BPE。BPE 会将稀有词拆分成更小的部分，从而构建出类似“语言元素周期表”的东西。例如，“aphr”是一个常见的英语前缀，所以 BPE 分词器可能会将“Aphra”分为两个标记：“aphr”和“a”。你可能还会发现，词表中的实际标记是“ aphr”和“a ”，因为 BPE 会像处理其他字符一样记录空格的位置信息。

BPE 提供了多语言的灵活性，也增强了系统对拼写错误和笔误的容错能力，比如能识别“aphradesiac”这种变体词。甚至像“African American”这样的少数群体 2-gram 也能在 BPE

的投票机制中拥有自己的表示。过去我们只能用 OOV 标记这一权宜之计来处理语言中的稀有现象，而现在不再需要这样做。正因为如此，最先进的深度学习 NLP 流水线（例如 Transformer）几乎都使用类似 BPE 的 **WordPiece 分词**方式。

BPE 通过字符和 WordPiece 标记组合，能为任何未知单词或词的一部分提供语义线索。例如，如果词表中没有“syzygy”，我们可以将其表示为六个字符标记：`s`、`y`、`z`、`y`、`g` 和 `y`。

或者，像“smartz”这种词可能被表示为两个标记：`smart` 和 `z`。

听起来确实很智能。下面我们就来看看它在实际文本语料中的效果。

```python
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(ngram_range=(1, 2), analyzer='char')
>>> vectorizer.fit(texts)
CountVectorizer(analyzer='char', ngram_range=(1, 2))
```

我们创建了一个 `CountVectorizer` 类，它将文本划分为字符（character），而不是单词（word）来进行分词（tokenize）。它不仅会统计单字符的 token，还会统计 token 对（二元字符组，character 2-grams），也就是 BPE（Byte Pair Encoding，字节对编码）中的字节对。现在我们可以检查我们的词汇表，看看它的样子：

```python
>>> bpevocab_list = [
... sorted((i, s) for s, i in vectorizer.vocabulary_.items())]
>>> bpevocab_dict = dict(bpevocab_list[0])
>>> list(bpevocab_dict.values())[:7]
[' ', ' a', ' c', ' f', ' h', ' i', ' l']
```

我们配置 `CountVectorizer` 将文本划分为所有可能出现的字符 1-gram 和字符 2-gram。`CountVectorizer` 会按字典顺序排列词汇，因此以空格字符（' '）开头的 n-gram 会排在最前面。一旦向量器知道了它需要统计哪些 token，它就可以将文本字符串转换为向量，每一个 token 在我们字符 n-gram 词汇表中占据一个维度：

```
>>> vectors = vectorizer.transform(texts)
>>> df = pd.DataFrame(
...     vectors.todense(),
...     columns=vectorizer.get_feature_names_out())
>>> df.index = [t[:8] + '...' for t in texts]
>>> df = df.T
>>> df['total'] = df.T.sum()
>>> df
Trust me... There's ... total
 t 31 14 45
 r 3 2 5
 u 1 0 1
 s 0 1 1
 3 0 3
.. ... ... ...
at 1 0 1
ma 2 1 3
yb 1 0 1
be 1 0 1
e. 0 1 1
<BLANKLINE>
[148 rows x 3 columns]
```

这个 DataFrame（数据帧）为每个句子创建了一列，为每个字符 2-gram 创建了一行。来看前四行，其中“r ”这个字节对（字符 2-gram）在这两个句子中分别出现了三到五次。所以即使是空格在构建 BPE 分词器时也会被算作字符。这正是 BPE 的优势之一：它会自动识别你的 token 分隔符，这意味着它也适用于那些词与词之间没有空格的语言。BPE 还可以用于替换密码文本（substitution cipher text），例如 ROT13，这是一种将字母表向前旋转 13 个字符的玩具加密法：

```
>>> df.sort_values('total').tail()
 Trust me... There's ... total
 en 10 3 13
 an 14 5 19
 uc 11 9 20
 e 18 8 26
 t 31 14 45
```

接下来，BPE 分词器会找到最常出现的 2-gram，并将它们加入永久词汇表。随着时间推移，它会删除那些频率较低的字符对，因为它们在之后的文本中再次出现的可能性越来越小：

```
>>> df['n'] = [len(tok) for tok in vectorizer.vocabulary_]
>>> df[df['n'] > 1].sort_values('total').tail()
 Trust me... There's ... total n
ur 8 4 12 2
en 10 3 13 2
an 14 5 19 2
uc 11 9 20 2
e 18 8 26 2
```

在 BPE 分词器的下一轮预处理（preprocessing）中，可能会保留 en、an，甚至 e 这样的字符 2-gram。然后，BPE 算法会用这个较小的字符 bigram（双字符组合）词汇表再次遍历文本，寻找这些字符 bigram 之间以及与单字符之间的高频组合。这个过程会持续进行，直到达到最大 token 数为止，并将最长的字符序列纳入词汇表中。

注意：你可能会听说 WordPiece（词片）分词器，例如 BERT 及其衍生模型所使用的，就是一种这种类型的分词器。这种分词器的工作原理与 BPE 相同，但它实际上会使用底层语言模型（language model）来预测字符串中相邻字符的概率。它会从词汇表中剔除那些对语言模型准确率影响最小的字符。数学处理上略有不同，并会产生略有差异的 token 词汇表，但你不需要主动选择这种分词器。使用它的模型会自动包含它，并集成在模型的处理流程中。

基于 BPE 的分词器所面临的一个主要挑战是：它们必须在你的特定语料库（corpus）上进行训练。因此，BPE 分词器通常只在 transformer（变换器）和 LLM（大型语言模型）中使用，你将在第 9 和第 10 章学习这些内容。

BPE 分词器的另一个挑战是你需要做很多记录来追踪每个已训练的模型所使用的分词器。Hugging Face 的一个重大创新就是专门解决了这个问题。他们的团队使得存储和共享所有预处理数据（例如分词器词汇表）变得简单，并且这些预处理数据可以和语言模型一起打包共享。这使得复用和共享 BPE 分词器变得更加容易。如果你希望成为一名自然语言处理专家，你也许可以仿照 Hugging Face 在 NLP 预处理流程上的做法来构建你自己的处理流水线。

到目前为止，你已经学习了几种将文本划分为标记的方法。这些 token 将帮助你构建词汇表，即文本中所有可能出现的标记的集合；但只是直接使用这些标记可能并不是最优的方法。在下一节中，你将学习几种提升词汇表质量的技术。

2.3 改进你的词汇表

到目前为止，你所使用的是构建词汇表最基础的方法：找出所有可能的标记，并将每一个标记都作为词汇表中的一个“词项（term）”。这种方法可以帮助你从文本中构建向量，但这些向量在分类（classification）或情感分析（sentiment analysis）等任务中的表现并不会太好，原因有很多。首先，把每个标记单独处理而忽略它们的共现关系（co-occurrence），会丢失大量信息。同样地，如果你不将本质上属于同一个词的不同形式连接起来，你的向量表示的表现也会下降。例如，如果包含 *swim* 和 *swimming* 的句子将这两个词表示为不同的标记，那么找到所有关于游泳的文章就会变得更加困难。所以我们接下来将探索一些可以帮助你提高词汇表效率的技术。

2.3.1 用 n-gram 扩展你的词汇表

让我们回顾一下本章开头提到的冰淇淋（ice cream）问题。还记得吗？我们当时讨论过如何把 *ice* 和 *cream* 绑在一起：

> I scream, you scream, we all scream for ice cream.
> 我尖叫，你尖叫，我们都为冰淇淋尖叫。

但我们认识的人中，很少有人会为 *cream* 尖叫。而且没有人会为 *ice* 尖叫，除非他们正要踩上去摔倒。所以，你需要一种方法让你的词向量（word vectors）把 *ice* 和 *cream* 绑在一起。

我们都为 N-gram 尖叫

一个n元语法 *（n-gram）* 是从一串元素中提取出的最多包含 *n* 个元素的序列，通常这些元素来自一个字符串。一般来说，n元语法的元素可以是字符、音节、词，甚至是符号（例如 DNA 或 RNA 序列中的化学氨基酸标记：A、D 和 G）。

在本书中，我们只关注 *词语级别* 的n元语法，而不是字符级别的。所以在这里，当我们说 *2-gram（2 元语法片段）*，我们是指一对词语，比如 *ice cream（冰淇淋）*。当我们说 *3-gram（3 元语法片段）*，我们指的是三个词组成的短语，比如 *beyond the pale（不合常规）*、*Johann Sebastian Bach（约翰·塞巴斯蒂安·巴赫）* 或 *riddle me this（猜猜这个）*。这些n元语法不一定非得组成固定搭配或复合词，只需要它们在文本中以足够频繁的形式共同出现，足以引起标记计数器的注意即可。

那为什么要使用 n-gram 呢？ 正如你之前看到的，当一个 token 序列被向量化为 *词袋模型（bag-of-words，BOW）* 向量时，它会丢失词语顺序中所固有的很多语义信息。通过将标记的定义扩展为包含多词的标记，也就是n元语法，你的NLP处理流程就能保留语句中词序本身所携带的更多含义。例如，具有否定语义的词 *not* 将能保留在它所属的上下文中，而不是“游离”在整个

句子或文档中。比如 2-gram *was not* 比起单独的 *was* 和 *not* 两个 1-gram 来说，保留了更多的上下文意义。在你的处理流程中将词与其相邻词组合，可以更好地保留其语义上下文。

在下一章中，我们会展示如何识别哪些 n-gram 相较其他更具有信息量，从而帮助你减少NLP流水线中需要维护的标记数量。否则系统将不得不维护每一个出现过的词序列的列表。n-gram 的优先排序将帮助处理流程正确识别出 *Three Body Problem（三体）* 和 *ice cream（冰淇淋）*，而不是混淆为 *three bodies（三个身体）* 或 *ice shattered（冰碎了）*。在第四章中，我们还会进一步将词对甚至更长的序列与其实际含义关联起来，而不再依赖其组成词的单独含义。但目前为止，你需要做的是让你的分词器能够生成这些词序列，即这些 n-gram。

停用词（Stop Words）

停用词（stop words） 是在任何语言中都很常见、出现频率很高但对短语语义贡献较少的词。以下是一些常见的停用词示例：

- a, an
- the, this
- and, or
- of, on

从历史上看，为了减少从文本中提取信息的计算开销，NLP流水线（NLP pipeline）中通常会排除这些停用词。尽管这些词本身信息量较少，但作为n元语法的一部分，停用词可以提供重要的关系信息。请看以下两个例子：

- Mark reported to the CEO
- Suzanne reported as the CEO to the board

在你的NLP流水线（NLP pipeline）中，可能会生成像“reported to the CEO”和“reported as the CEO”这样的 4-gram（四元组）。如果从这些 4-gram 中移除了停用词，两个例子都将被简化为“reported CEO”，从而丢失了关于职务等级的关键信息。在第一个例子中，Mark 可能是向 CEO 汇报的助理，而在第二个例子中，Suzanne 本身是 CEO，向董事会汇报。

然而，在流程中保留停用词也会带来另一个问题：为了利用由这些本身无意义的停用词所建立的连接关系，你需要更长的 n-gram。这意味着，如果想避免人力资源示例中的歧义，至少需要保留 4-gram。

为停用词设计筛选器，需要依据具体应用场景进行。词汇量将直接影响NLP流水线后续所有步骤的计算复杂度和内存需求，而停用词仅占整个词汇量的一小部分。一个典型的停用词表通常只包含大

约 100 个高频但不重要的词。另一方面，要追踪在大型语料库（如推文、博客和新闻文章）中出现的 95% 的词，仅单词级别（1-gram）就可能需要 20,000 个词汇。要覆盖大型英文语料库中 95% 的 2-gram，通常需要超过 100 万个独特的 2-gram 标记。

你可能会担心词汇量越大，所需的训练集规模也越大，以避免对某些词或词组的过拟合。而训练集的大小也决定了处理该训练集所需的资源。然而，从 20,000 个词中去掉 100 个停用词，对提升效率的影响微乎其微。而在 2-gram 词汇表中，若不加选择地删除停用词，反而会损失大量信息。例如，你可能无法识别“The Shining”作为一个独特的片名，而把它和包含“Shining Light”或“shoe shining”的文档等同处理。

因此，如果你的系统具备足够的内存和计算能力来处理更大的词汇表，就没有必要过分担心保留一些无关紧要的词。而如果你担心词汇量太大导致在小训练集上出现过拟合问题，那么还有更好的方法来选择词汇或降低维度，而不必单纯依靠删除停用词来实现。将停用词纳入词汇表，还能让文档频率过滤器（详见第3章）更准确地识别并忽略在你的特定语料中信息含量最低的词和n元语法。

代码清单 2.2：一个广泛的停用词列表 Python

```
>>> import requests

# 这个详尽的停用词（stop words）列表汇总自 SEO 列表、spaCy 和 NLTK
>>> url = ("https://gitlab.com/tangibleai/nlpia/-/raw/master/"
... "src/nlpia/data/stopword_lists.json")
>>> response = requests.get(url)
>>> stopwords = response.json()['exhaustive']

# 为了简化本示例，句子中的标点符号和大小写已被移除
>>> tokens = 'the words were just as I remembered them'.split()
>>> tokens_without_stopwords = [x for x in tokens if x not in stopwords]
>>> print(tokens_without_stopwords)
['I', 'remembered']
```

这是泰德·姜（Ted Chiang）的一篇短篇小说中的一句有意义的句子，故事讲述了机器帮助我们记住曾经说过的话语，从而无需依赖不可靠的记忆系统。

在这个短语中，尽管你保留了部分句意，但丢失了三分之二的词汇；而你也可以看到，通过使用这样一组特别详尽的停用词列表，一些重要的标记被舍弃了。有时候，你也许可以在不使用冠词、介词，甚至不使用动词“to be”的各种变形的情况下表达清楚自己的意思，但这将降低NLP流水线的精度和准确性，并且至少会造成一部分语义信息的丢失。

你可以观察到，不同词语所承载的语义权重是不同的。想象一个正在使用手语交流的人，或是一个急匆匆记下便条的人。他们在赶时间时会选择跳过哪些词？语言学家正是通过这种方式来确定哪些词应当列入停用词列表的。但如果你自己正在赶时间，而你的NLP流水线并不需要“赶时间”，那么你大可不必耗费精力去手动创建和维护停用词列表。

以下代码清单展示了另一组广泛使用的常见停用词列表，虽然不如前面提到的那组那么详尽。

代码清单 2.3：NLTK停用词列表 Python

```
>>> import nltk
>>> nltk.download('stopwords')
>>> stop_words = nltk.corpus.stopwords.words('english')
>>> len(stop_words)
179
>>> stop_words[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
>>> [sw for sw in stopwords if len(sw) == 1]
['i', 'a', 's', 't', 'd', 'm', 'o', 'y']
```

一份过于侧重第一人称的文档通常显得相当乏味，更重要的是，它的信息含量很低。NLTK 包在其停用词（stop words）列表中包含了代词（不仅仅是第一人称的）。这些单字符的停用词可能看起来有些奇怪，但如果你经常使用 NLTK 的分词器（tokenizer）和 Porter 词干提取器（stemmer），它们的存在就很容易理解了。当使用 NLTK 的分词和词干提取工具对缩略词进行拆分和词干化时，这些单字符的标记经常会出现。

⚠ **警告**：Scikit-learn、spaCy、NLTK 和 SEO 工具中使用的英文停用词集合差异很大，且它们都在持续演变之中。截至本书撰写时，Scikit-learn 的停用词有 318 个，NLTK 有 179 个，spaCy 有 326 个，而我们所采用的“详尽”SEO 列表包含 667 个停用词。这也是为什么你应当认真考虑是否需要对停用词进行过滤的一个充分理由。如果你选择过滤停用词，其他人可能就无法复现你的实验结果。

2.3.2 规范化你的词汇表（Normalizing your vocabulary）

你已经了解了词汇表大小对NLP流水线性能的重要性。另一种“清理”词汇表的技巧是对词汇表进行规范化，即将含义相近的标记合并为一个规范化的形式。这样可以减少需要在词汇表中保留的标记数量，同时还能提升你在语料中不同“拼写”形式之间的语义关联能力。正如我们之前提到的，减少词汇表大小还可以降低模型过拟合的风险。

大小写归一化（CASE FOLDING）

大小写归一化（case folding）指的是将仅在大小写上不同的多个“拼写”形式统一为一个形式。那我们为什么需要大小写归一化呢？词语的大小写常常因出现在句首或为了强调而被全部大写，导致词语变得“大小写非标准化”。

对这种“非标准化”进行修正的过程称为大小写标准化（case normalization），或更常见的称法是**大小写归一化（case folding）**。对词和字符的大小写进行归一化是减少词汇表大小、提升NLP流水线泛化能力的一种方式。它可以帮助你将本意相同、拼写应一致的词汇合并为一个统一的标记。

然而，需要注意的是，大小写有时也传递了关键信息——例如，"doctor" 和 "Doctor" 往往意义不同。大小写常被用来标识专有名词，如人名、地名或事物名。如果你的NLP流水线中需要进行**命名实体识别（Named Entity Recognition, NER）**，你可能会希望将专有名词与普通词区别对待。

不过，如果标记没有进行大小写归一化，你的词汇表大小可能会翻倍，占用两倍的内存和处理时间，也会显著增加训练集所需标注的数据量，从而使机器学习流水线难以收敛到一个准确且具有泛化能力的解。就像其他机器学习任务一样，你用于训练的有标签数据必须能“代表”所有可能的特征向量空间，这其中也包括大小写的变体。对于维度为 100,000 的词袋模型向量，你通常需要 100,000 条有标签样本，甚至更多，才能训练出不过拟合的监督式机器学习模型。在某些场景下，将词汇表大小减半所带来的信息损失，可能是可以接受的代价。

在 Python 中，你可以使用列表推导式轻松地对标记的大小写进行归一化处理：

```python
>>> tokens = ['House', 'Visitor', 'Center']
>>> normalized_tokens = [x.lower() for x in tokens]
>>> print(normalized_tokens)
['house', 'visitor', 'center']
```

如果你确信自己想要对整个文档进行大小写归一化处理，可以在分词前通过 `.lower()` 方法一次性将文本字符串转换为小写。但这种方式会阻碍那些能够识别驼峰命名（CamelCase）词语的高级分词器的使用，比如 WordPerfect、FedEx 或 stringVariableName。或许你希望将 WordPerfect 作为一个独立的标记，又或者你只是想回忆起那个更加完美的文字处理时代。是否执行大小写归一化、何时执行，完全取决于你自己。

通过大小写归一化处理，你的目标是将标记还原到它们的“常态”形式——即尚未受到语法规则或句中位置影响而发生大小写变化的状态。对文本进行大小写归一化最简单、最常见的方法就是使用 Python 内置的 `str.lower()` 函数将所有字符转换为小写。

但不幸的是，这种方式除了去除你本意上要规避的句首大写（即句子首词因语法规则被强制大写）之外，也会一并“归一化”掉很多具有实际语义的大小写信息。更好的方法是**只将每个句子的第一个词转为小写**，而保留其余词语的原始大小写形式。

仅对句首词进行小写处理可以保留句中专有名词的语义，例如 "Joe" 和 "Smith" 在 "Joe Smith" 中的用法。同时，它还能让那些只在句首才大写的普通词合理归组。这样就能避免将 "Joe" 与“咖啡”（即“joe”）混淆，也避免将“smith”作为工匠的通用称谓与“Smith”这一姓氏混为一谈。比如在句子 "A word smith had a cup of joe" 中，标记 "smith" 是普通名词，而 "Joe Smith, the word smith, with a cup of joe" 中的 "Smith" 是专有名词。即使你采用了这种更谨慎的大小写归一化方法（即只小写句首词），你仍可能因句首出现的专有名词而引入错误。例如，句子“Joe Smith, the word smith, with a cup of joe.”所产生的标记集合将与“Smith the word with a cup of joe, Joe Smith.”不同，而你可能并不希望出现这种差异。此外，对于没有大小写区分概念的语言（如阿拉伯语或印地语），大小写归一化也毫无意义。

为避免信息丢失，很多NLP流水线完全不进行大小写归一化。对于许多应用场景而言，通过减少词汇表规模约一半所带来的效率提升（在存储与处理方面），往往不足以抵消专有名词语义丢失的代价。当然，即便你不进行大小写归一化，信息丢失仍有可能发生。例如，将句首的“The”误当作有意义词语而未将其识别为停用词，在某些任务中也会带来困扰。更为复杂的NLP流水线会在执行大小写归一化前先识别出专有名词，然后仅对句首那些明显不是专有名词的词执行大小写归一化。你应根据自身应用场景选择合适的大小写归一化策略。如果你的语料中并没有大量的“Smiths”与“word smiths”，而你也不介意它们是否被归为同一标记，那么你完全可以简单地把所有文本转换为小写。最好的方式，是尝试多种不同的归一化策略，并选择在你的自然语言处理项目目标下表现最好的方法。

通过使你的模型能处理各种奇特大小写方式的文本，大小写归一化有助于降低机器学习流程中的过拟合风险。对于搜索引擎而言，大小写归一化尤其实用。它可以提升查询匹配的数量，这种提升通常称为召回率（recall），这是搜索引擎（或其他分类模型）中的一个性能指标。

假设你使用一个未进行大小写归一化的搜索引擎，若你搜索“Age”，返回的文档会与搜索“age”时的结果完全不同。前者可能出现在诸如“New Age”或“Age of Reason”的短语中，而后者更可能出现在类似“at the age of”这样的句子中（比如关于托马斯·杰斐逊的描述）。通过对搜索索引（以及用户查询）进行词汇归一化处理，你可以确保无论用户在查询中使用的是哪种大小写，两种类型的文档都能被一并返回。

这种更高的召回率是以牺牲精确度（precision）为代价的，即用户可能会收到更多他们不感兴趣的结果。因此，现代搜索引擎通常允许用户在每次查询时关闭大小写归一化，通常方式是给想要精

确匹配的词语加引号。如果你正在构建这样的搜索引擎流程，那么为了支持这两种类型的查询，你需要为文档构建两个索引：一个使用大小写归一化的n元语法索引，另一个保留原始大小写。

词干提取（Stemming）

另一种常见的词汇归一化技术是消除复数、所有格词尾，甚至不同动词时态之间所带来的细微语义差异。这种归一化方法通过识别词语之间的共用词干来实现，称为**词干提取（stemming）**。例如，单词 *housing* 和 *houses* 拥有相同的词干：*house*。词干提取的目标是通过去除单词的后缀，将语义相近的词语合并到一个共有的词干下。词干本身不一定是拼写正确的词语，它只是一种标记（token）或标签（label），代表某一组可能的词语拼写形式。

人类很容易看出 *house* 与 *houses* 是同一个名词的单数与复数形式，但你需要某种方式将这种信息传递给机器。词干提取的一个主要优势在于它能够压缩需要被语言模型或程序处理的词汇量。它能减少词汇表的规模，同时尽量降低语义信息的损失。在机器学习中，这种处理被称为维度约简（dimension reduction）。它有助于使语言模型泛化：凡是被归入相同词干索引位置的词语，模型都能对其作出一致的响应。只要你的应用场景不要求区分 *house* 和 *houses*，那么这个词干就能将程序的代码量或数据集规模减少一半，甚至更多——具体取决于你选择的词干提取器的“激进程度”。

词干提取在关键词搜索或信息检索中非常重要。它允许你在搜索“developing houses in Portland”时，找到使用了 *house*、*houses*，甚至 *housing* 的网页或文档——因为这些词在提取词干时都会合并成 *hous*。同理，你可能还会检索到包含 *developer* 或 *development* 的页面，而不仅仅是含 *developing* 的内容，因为这些词都通常被归约为词干 *develop*。正如你所看到的，这种处理方式会拓宽搜索范围，确保你不太可能遗漏掉某个相关的网页或文档。

这种搜索范围的拓展会显著提升搜索引擎的召回率（recall），也就是衡量搜索引擎是否能返回所有相关文档的指标。然而，词干提取也可能显著降低搜索引擎的精确率（precision），因为它可能会返回很多不相关的页面，与相关内容混杂在一起。在某些应用中，这种误报率（false-positive rate）（即被返回但用户并不觉得有用的页面比例）会成为一个问题。

因此，大多数搜索引擎允许用户**关闭词干提取和大小写归一化**，只需在搜索词或短语上**加上引号**即可。加引号表示你只想检索拼写**完全匹配**的页面，例如搜索："Portland Housing Development software"

这样的搜索将返回与："a Portland software developer’s house"完全不同类型的文档。在某些情况下，你可能想要搜索的是“Dr. House’s calls”，而不是“dr house call”。如果你对查询应用了词干提取，后者可能就变成了实际的查询内容，从而**失去了原始语义**。

下面是一个用纯 Python 编写的简易词干提取器，它可以处理结尾为 `s` 的情况：

```python
>>> def stem(phrase):
... return ' '.join([re.findall('^(.*ss|.*?)(s)?$',
... word)[0][0].strip("'") for word in phrase.lower().split()])
>>> stem('houses')
'house'
>>> stem("Doctor House's calls")
'doctor house call'
```

前面的词干提取函数遵循了几个简单的规则，这些规则都被封装在一个简洁的正则表达式中：

- 如果一个词**以多个字母 s 结尾**，那么词干（stem）就是整个单词，后缀（suffix）为空字符串；
- 如果一个词**只以一个 s 结尾**，则词干是去掉结尾的 s 后的词，后缀就是这个 s；
- 如果一个词**不以 s 结尾**，则词干为原词，不返回后缀。

此外，使用 `strip` 方法还可以使得一些**所有格形式的单词**（如 *teacher's*）与**复数形式的词**（如 *teachers*）一并被处理。

这个函数在处理**常规情况**时效果良好，但对一些**更复杂的情况**则无能为力。例如，它无法正确处理诸如 *dishes* 或 *heroes* 这类单词。这些单词由于词尾变化规律复杂，简单的规则难以涵盖。类似 *housing* 这样的例子（比如出现在你用于搜索“Portland Housing”的场景中），这个函数同样处理不了。

为了解决这些复杂的英语词尾变化问题，**NLTK** 包提供了更强大的词干提取器。其中最常用的两种词干提取算法是：

- **Porter 词干提取器（Porter stemmer）**
- **Snowball 词干提取器（Snowball stemmer）**

Porter 词干提取器以计算机科学家 **Martin Porter** 的名字命名。他在 1980 年代和 1990 年代致力于完善这一算法。Porter 还在其基础上进一步改进，提出了 Snowball 词干提取器，又称为 Porter2 算法。Porter 几乎把自己的整个职业生涯都奉献给了词干提取算法的研究与改进，尤其在信息检索（如关键词搜索）领域，这些工具具有极高的价值。与我们前面实现的那个正则表达式方法相比，Porter 和 Snowball 提取器实现了更复杂的词尾处理规则，能够更好地处理英语中的拼写变化和构词规则。

```Python
>>> from nltk.stem.porter import PorterStemmer
>>> stemmer = PorterStemmer()
>>> ' '.join([stemmer.stem(w).strip("'") for w in
... "dish washer's fairly washed dishes".split()])
'dish washer fairli wash dish'
```

Porter 词干提取器与我们之前提到的正则表达式版本类似，也会保留尾部的撇号（’），除非你明确地将其去除。这个特性确保了所有格形式的单词（如 *teacher's*）仍然可以和非所有格形式（如 *teachers*）加以区分。所有格词通常是专有名词，因此在某些需要对人名进行特殊处理的应用中，这种差别是非常重要的。

关于 Porter 词干提取器的更多信息

Julia Menchavez 慷慨地公开了她用纯 Python 语言翻译的 **Porter 原始词干提取算法实现**：

🔗 GitHub 地址

如果你曾考虑自己动手开发一个词干提取器，那么这份代码可以让你**直观感受到 Porter 所投入的 300 多行代码和数十年算法调优的深度**。
Porter 提取器并非简单地删减词尾，它通过多个步骤，逐步将单词简化为其“基本词干”。

Porter 词干提取算法的八个步骤如下：

- **Step 1a**：处理以 s 结尾的词（与前面正则表达式方法类似），例如处理单复数词尾。

```Python
def step1a(self, word):
 if word.endswith('sses'):
  word = self.replace(word, 'sses', 'ss')
  elif word.endswith('ies'):
  word = self.replace(word, 'ies', 'i')
  elif word.endswith('ss'):
  word = self.replace(word, 'ss', 'ss')
  elif word.endswith('s'):
  word = self.replace(word, 's', '')
  return word
```

剩下的七个步骤要复杂得多，因为它们需要处理英语拼写规则中的复杂情况：

- **步骤 1a** — 以 s 和 es 结尾的词
- **步骤 1b** — 以 ed、ing 和 at 结尾的词
- **步骤 1c** — 以 y 结尾的词
- **步骤 2** — 名词化后缀，例如 ational、tional、ence 和 able
- **步骤 3** — 形容词后缀，例如 icate、ful 和 alize
- **步骤 4** — 形容词和名词后缀，例如 ive、ible、ent 和 ism
- **步骤 5a** — 任何剩余的顽固 e 结尾
- **步骤 5b** — 以双辅音结尾的词干将以单个 l 结尾

Snowball 词干器比 Porter 词干器更加激进。请注意，它会将 fairly 词干化为 fair，这比 Porter 更准确。

Python

```
>>> from nltk.stem.snowball import SnowballStemmer
>>> stemmer = SnowballStemmer(language='english')
>>> ' '.join([stemmer.stem(w).strip("'") for w in
... "dish washer's fairly washed dishes".split()])
'dish washer fair wash dish'
```

词形还原（Lemmatization）

如果你拥有关于不同单词之间意义联系的信息，你就有可能将几个拼写不同的单词关联在一起。这种更深入地归一化到单词语义根（即标记）的过程被称为**词形还原（lemmatization）**。

在第 3 章和第 11 章中，我们将展示如何使用词形还原来降低聊天机器人对话响应逻辑的复杂性。任何希望对同一基本词根的不同拼写做出相同反应的NLP流水线都能从词形还原器中受益。它减少了你需要响应的词数——即语言模型的维度。词形还原可以让你的模型更具泛化能力，但也可能降低模型的精确性，因为它会将所有来自同一词根的拼写变体视为同一个词。例如，chat、chatter、chatty、chatting，甚至 chatbot，都会在使用词形还原的NLP流水线中被视为相同，尽管它们的含义不同。同样地，bank、banked 和 banking 在词干提取流程中也会被视为相同，尽管它们可能指的是河岸（bank）、倾斜（banked）、或银行业务（banking）等不同含义。

在阅读本节内容时，思考一些在词形还原后会极大改变其原意的词，甚至可能颠倒原本含义，这种情况可能会使你的流程做出错误反应。这种情况被称为**欺骗攻击（spoofing）**，即通过精心构造

具有挑战性的输入诱导机器学习流程做出错误响应。

有时候，词形还原是归一化词汇的更好方式。你可能会发现，对于你的应用来说，词干提取和大小写统一会将一些本不该归并的词合并成同一个词干和标记。而词形还原器利用单词的同义词和词尾的知识库，确保只有意义相近的单词才会被归并到一个统一的标记上。

有些词形还原器还会结合词性标签（POS tag）以提高准确性。一个词的词性标签表示它在短语或句子中语法上的角色。例如，名词词性表示人、地点或事物；形容词词性表示对名词的修饰或描述；动词词性表示动作。要确定一个词的词性，就必须了解它在上下文中的使用方式；因此，一些高级词形还原器不能单独处理一个个孤立的词。

你能想到利用词性信息来识别一个比词干提取更好的“词根”的方法吗？思考一下单词 **better**。词干提取器会去掉其后缀 er，返回的词干可能是 **bett** 或 **bet**。然而，这样会把 better 和 betting、bets 以及 Bet’s 等词归为一类，而不是和意义更接近的词，比如 betterment、best 甚至 good 和 goods。

因此，**在大多数应用中，词形还原比词干提取更优**。词干提取器主要用于大规模的信息检索应用（例如关键词搜索）。如果你确实希望在信息检索流程中获得词干提取带来的维度缩减和召回率提升，那么你也应该在词干提取之前使用词形还原器。因为标记是合法的英文单词，词干提取器对词形还原器的输出处理效果更好。这个小技巧能进一步降低维度，并比单独使用词干提取器带来更高的信息检索召回率。

那么，如何在 Python 中识别单词的标记呢？NLTK 包提供了相关函数。请注意，为了获得最准确的标记，你需要告诉 `WordNetLemmatizer` 你感兴趣的词性（POS）：

```
>>> nltk.download('wordnet')
True
>>> nltk.download('omw-1.4')
True
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("better") # 默认的pos参数是n(名词)
'better'
>>> lemmatizer.lemmatize("better", pos="a") # "a"表示形容词(adjective)
'good'
>>> lemmatizer.lemmatize("good", pos="a")
'good'
>>> lemmatizer.lemmatize("goods", pos="a")
'goods'
>>> lemmatizer.lemmatize("goods", pos="n")
'good'
>>> lemmatizer.lemmatize("goodness", pos="n")
'goodness'
>>> lemmatizer.lemmatize("best", pos="a")
'best'
```

你可能会惊讶地发现，第一次尝试对单词 **better** 进行词形还原时，它并没有发生任何变化。这是因为一个词的词性对其含义有重大影响。如果没有为某个单词指定词性，NLTK 的词形还原器会默认它是一个名词。一旦你为它指定了正确的词性（在这个例子中是形容词 **adjective**），词形还原器就会返回正确的标记。

不过，可惜的是，NLTK 的词形还原器仅限于普林斯顿大学 WordNet 词义图中的词汇连接。因此，“best”这个词无法还原到与“better”相同的词根。同时，该图谱中也缺少“goodness”与“good”之间的联系。相比之下，**Porter 词干提取器** 会盲目地去掉所有单词的 **–ness** 后缀，从而将这些词连接起来:

```
>>> stemmer.stem('goodness')
'good'
```

在 spaCy 中实现词形还原则很简单：

```python
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")
>>> doc = nlp("better good goods goodness best")
>>> for token in doc:
>>> print(token.text, token.lemma_)
better well
good good
goods good
goodness goodness
best good
```

与 NLTK 不同，spaCy 会将“better”当作 **副词（adverb）** 来处理，并将其标记正确还原为“well”；对于“best”，它也能返回正确的标记“good”。

同义词替换（SYNONYM SUBSTITUTION）

有五种类型的同义词替换在创建一个更一致、更精简的词汇表时可能很有用，有助于你的 NLP 流水线实现良好的泛化能力：

- 打字错误更正
- 拼写修正
- 同义词替换
- 缩写还原
- 表情符号扩展

每种同义词替换算法都可以设计得更激进或更保守，你需要根据用户在特定领域中的语言使用习惯进行权衡。例如，在法律、技术或医学等专业领域中，替换同义词通常不是个好主意。医生可不希望聊天机器人因为替换了心形表情符号（<3）而告诉病人“你的心碎了”。尽管如此，词形还原和词干提取的适用场景同样适用于同义词替换。

词形还原、词干提取与同义词替换的使用场景

什么时候应使用词形还原、词干提取或同义词替换？词干提取的计算速度通常更快，代码和数据集的复杂度也较低，但词干提取错误更多，会合并更多不同的单词，从而比词形还原更大程度地降低文本的信息含量或语义。词干提取和词形还原都会减少词汇表的大小，并增加文本的歧义性，但词

形还原能更好地保留文本中单词的原始含义和上下文信息。因此，一些先进的 NLP 工具包（如 spaCy）不再提供词干提取功能，而只提供词形还原方法。

如果你的应用涉及搜索，词干提取和词形还原可以通过将更多文档与相同查询词相关联，从而提升搜索的召回率。不过，这些词汇压缩方法（包括大小写标准化）通常会降低搜索结果的精准率和准确度，因为它们可能导致搜索系统返回许多与原始词义无关的文档，也就是误报。有时，与其担心误报，不如担心漏报（即搜索引擎根本没有找到你要找的文档）。

因为搜索结果可以按照相关性进行排序，搜索引擎和文档索引系统通常在处理你的查询和索引文档时使用词形还原。这意味着搜索引擎会在对你的查询文本进行分词时使用词形还原，也会在处理它所抓取的网页等文档时使用它。不过，搜索引擎会结合未经过词干提取的词语来排序它最终呈现给你的搜索结果。

对于一个基于搜索的聊天机器人来说，精准率通常比召回率更重要。误报可能导致聊天机器人说出不恰当的话，而漏报最多只会让机器人谦虚地承认自己找不到合适的回答。若你的聊天机器人希望表达得更自然可信，建议 NLP 流水线优先使用未经词干提取或词形还原的原始词语进行匹配。若未能匹配成功，再退而求其次地使用经过标准化处理的词语匹配。同时，你可以将这些退而求其次的结果排在较低的优先级。

你甚至可以让你的机器人表现出谦逊和透明，用提示语引入这些低优先级的答案，例如：“我以前没听过类似的说法，但使用我的词干提取器后，我找到了这样的内容……”。在这个充满“夸夸其谈”型机器人的现代社会，你的谦逊型机器人反而可能脱颖而出，赢得用户的喜爱！

在以下四种场景中，使用某种形式的**同义词替换**可能是合理的：

- 搜索引擎
- 数据增强
- 评估 NLP 模型的鲁棒性
- 对抗性 NLP

搜索引擎可以通过同义词替换来提高对稀有词的召回率。当你拥有的标注数据较少时，仅通过同义词替换，往往就可以将数据集扩展 10 倍。如果你想评估模型准确率的下界，可以在测试集上激进地进行同义词替换，从而观察你的模型对这些变化的鲁棒性。如果你正在寻找投毒模型或规避 NLP 检测的方法，同义词也能为你提供大量可以尝试的测试文本。例如，将“现金”（cash）、“美元”（dollars）或“格里夫纳”（乌克兰货币 hryvnia）互相替换，可能有助于绕过垃圾邮件检测器。

最终建议是：**除非你的文本非常有限，并且只关注特定词语的使用方式和大小写，否则尽量避免使用词干提取、词形还原、大小写标准化或同义词替换**。随着 NLP 数据集的爆炸式增长，这种需求

在英文文本中已非常罕见，除非你的文档中包含大量术语，或来自某个极小的科技、文学、或科学子领域。

尽管如此，对于非英文语言，你可能仍然会发现词形还原具有实际用途。斯坦福的信息检索课程甚至完全摒弃了词干提取和词形还原，理由是：这两种方法对召回率的提升可以忽略不计，反而会显著降低搜索的精准率。

2.4 具有挑战性的标记：处理表意文字语言

中文、日语以及其他表意文字语言并不像拼音文字语言（如英文）那样，只由少量字母组成标记或单词。这些语言中的字符更像图画，被称为**表意文字**（logographs）。中文中有成千上万个独特的字符，而这些字符的使用方式与我们在拼音语言中使用单词类似。但中文的每个字符通常不是一个完整的词；其含义依赖于其左右的其他字符，并且词语之间没有空格进行分隔。这使得将中文文本切分成词语或其他有意义的信息单位变得具有挑战性。

Jieba是一个用于将中文文本分词的 Python 包。它支持三种分词模式：

- 全模式 —— 提取句子中所有可能的词（用于召回）
- 精确模式 —— 精准地将句子切分成最合理的词语（用于理解）
- 搜索引擎模式 —— 将长词进一步切分成更短的词（就像在英文中拆分复合词或寻找词根）

以下例子中的中文句子：

> “西安是一座举世闻名的文化古城”

可以翻译为：“Xi’an is a city famous worldwide for its ancient culture.”
或者更简洁直译一些：“Xi’an is a world-famous city for her ancient culture.”

从语法角度看，这句话可以切分为以下几个部分：

- 西安（Xi’an）
- 是（is）
- 一座（a）
- 举世闻名（world-famous）
- 的（形容词后缀）
- 文化（culture）
- 古城（ancient city）

其中，字符“座”是**量词**，通常用于修饰“城市”这样的名词。Jieba 的**精确模式**会将句子切分成这种形式，以便你可以准确地提取出文本的语义含义。

代码清单2.4：精确模式下的 Jieba Python

```
>>> import jieba

# Jieba 的默认模式为精确模式。
>>> seg_list = jieba.cut("西安是一座举世闻名的文化古城")
>>> list(seg_list)
['西安', '是', '一座', '举世闻名', '的', '文化', '古城']
```

Jieba 的精确模式会最小化分词的总数，因为它尽可能将更多的字符组合在一起。这有助于减少错误将词语边界划分开的情况（即降低假阳性或第一类错误率）。如下一个代码清单所示，在全模式下，Jieba 会尝试将文本划分为更多更小的词语，因而分词数量会更多。

代码清单 2.5 全模式下的 Jieba Python

```
>>> import jieba

# cut_all=True 指定为全模式
>>> seg_list = jieba.cut("西安是一座举世闻名的文化古城", cut_all=True)
>>> list(seg_list)
['西安', '是', '一座', '举世', '举世闻名', '闻名', '的', '文化', '古城']
```

你可以在 https://github.com/fxsjy/jieba 找到关于 Jieba 的更多信息。SpaCy 也包含用于中文的语言模型，可以对中文文本进行分词和词性标注，例如 `zh_core_web_sm`。Jieba 包同样支持词性标注功能，但其词性标注模块并不支持 Python 的新版本（3.5 以上）。尽管如此，由于其易用性，Jieba 在近几年仍然十分流行，尽管它已经有一段时间没有维护了。

2.4.1 一个复杂的图景：中文中的词形还原与词干提取

与英语不同，象形文字语言（如中文和日语中的汉字）并没有词干提取（stemming）或词形还原（lemmatization）的概念。不过，倒是有一个相关的概念。中文字符的最基本构建单元是**部首（radicals）**。为了更好地理解部首，必须首先了解汉字是如何构成的。

汉字有六种类型，但其中最常见的四类最为重要，涵盖了大多数的中文字符：

- **象形字（Pictographs）** —— 由真实物体的图像构造而成的字符，例如“口”（mouth）和“门”（door）。
- **形声字（Phono-semantic compounds）** —— 由部首和一个汉字组成，例如“妈”（mā，mother）=“女”（female）+“马”（mǎ，horse）。
- **会意字（Associative compounds）** —— 由其他象形字符组成的汉字，例如“旦”（dawn），其上部分“日”表示太阳，下部分“一”象征地平线。
- **指事字（Self-explanatory characters or indicatives）** —— 难以通过图像直观表示的字符，因此用一个抽象符号表示，例如“上”（up）和“下”（down）。

如上所示，词干提取和词形还原在中文中更为复杂甚至不可行。将一个字符拆分开可能会显著改变其含义，而且部首的组合也没有统一规则或顺序。

不过，某些类型的词干提取在中文中反而比在英语中更简单。例如，从英语单词 *we, us, they, and them* 中自动去除复数形式是困难的；但在中文中则比较直接。中文使用后缀字符“们”来构造复数形式，类似于英语中加上“s”，如“朋友”（friend）变成“朋友们”（friends）。

即使是 *we/us, they/them, and y'all*，中文也使用相同的复数后缀，如“我们”（we/us）、“他们”（they/them）、“你们”（you）。而在英语中，你可以从动词中移除“ing”或“ed”来获得词根；但在中文中，动词变化通常通过在词首或词尾添加字符来表示时态。例如，“学”（learn）可以出现在“在学”（learning）和“学过”（learned）中。大多数情况下，你应保留该汉字整体，而非将其拆分为构成部分。

对于所有语言来说，一个很好的经验法则是：**如果统计结果表明词干提取或词形还原可以帮助 NLP 流水线表现更好，就使用它；否则不要使用。** 毕竟，smart、smarter 与 smartest 之间并不是没有语义差异。如果强行简化，会让 NLP 系统变“笨”。

让字符和词的统计学来指导你是否该将某个单词或 n-gram 分解。

2.5 标记向量（Vectors of tokens）

在你已经将文本切分为标记之后，接下来该怎么处理它们？你该如何将它们转化为对机器有意义的数字？

最简单的方式是检测你关心的某个标记是否出现。你可以通过硬编码逻辑来检查重要标记是否存在，这种方法称为**关键词（keywords）**检测。

这对于你在第 1 章中设计的问候意图识别器（如检测字符串开头的 *Hi* 和 *Hello*）效果不错。在新的分词文本中，你可以更好地检测到 *Hi* 和 *Hello* 的出现与否，而不会因为 *Hiking* 和 *Hell* 等词而产生混淆。有了新的分词器，你的 NLP 流水线就不会将 *Hiking* 错误地解析为 *Hi king*。

分词还能帮助你降低简单意图识别流程中的假阳性率。该流程试图识别问候词的存在，这种方法常被称为**关键词检测（keyword detection）**，因为你只关注有限词汇集合中的关键词。不过，如果你想覆盖问候语中可能出现的所有形式，包括俚语、拼写错误等，这种方法就显得很繁琐。你得写一个循环来枚举所有词，这效率很低。

我们可以借助线性代数和 `numpy` 的向量化运算来提高处理效率。因此你需要掌握一些代数知识，以便回答某个特定标记或“意图”是否出现在文本中。你将首先学习最基本、最直接、无损的表示标记为矩阵的方式：**独热编码（one-hot encoding）**。

2.5.1 独热向量（One-hot vectors）

现在你已经成功地将文档切分为各类标记，就可以将它们转化为向量。数字向量是我们进行自然语言处理数学计算的基础：

Python

```
>>> import pandas as pd
>>> onehot_vectors = np.zeros(
...     (len(tokens), vocab_size), int)  # 创建一个全为0的矩阵，行数为tokens的长度，列数为词汇表大小
>>> for i, tok in enumerate(tokens):
...     if tok not in vocab:
...         continue
...     onehot_vectors[i, vocab.index(tok)] = 1  # 对于出现在词汇表中的token，设置对应列为1
>>> df_onehot = pd.DataFrame(onehot_vectors, columns=vocab)
>>> df_onehot.shape
(18, 15)
>>> df_onehot.iloc[:, :8].replace(0, '') # 为简洁起见，这里只展示了 DataFrame 的前八列，并将 0 替换为了 ''
```

Python

```
# 结果待输出
```

在这个包含两句话引用的表示方法中，每一行是文档中某个词的一种向量表示。这个表格有15列，是因为词汇表中有15个独特的词。表格共有18行，对应文档中的每个词。某一列中的数字1表示该词出现在文档中的对应位置。

你可以从上到下“读取”这个独热编码（vectorized）的文本。你能看出文本中的第一个词是“There’s”，因为第一行中“There’s”所在列是1。接下来的三行（行索引1、2、3）是空白的，因为我们为了排版将表格右边的部分截断了。第五行（索引为4）显示的是文本中的第五个词是“adequate”，因为在对应列中有一个1。

独热向量是非常稀疏的，每一行向量中只包含一个非零值。为了显示方便，这段代码将0替换为空字符串（''），但这并不会实际更改你在NLP流水线中处理的 `DataFrame` 数据，仅仅是为了更易读。

注意 不要在你打算用于机器学习流水线的任何 `DataFrame` 中添加字符串。分词器和向量器（比如这种独热向量器）的目的是生成一个数值数组，便于你的NLP流水线进行数学运算。你无法对字符串进行数学运算。

表格中的每一行是一个二进制行向量，这也是它被称为**独热向量（one-hot vector）**的原因：除了一个位置（列）是1之外，其余都是0或空白。1（即“热”）表示存在，而0表示不存在。

这种词的向量表示和文档的表格表示的优点之一是信息不会丢失。表格中独热向量的排列顺序就编码了token的顺序。只要你知道哪个词由哪一列表示，你就可以从这个独热向量表格中完美地重建出原始的tokens序列。这种重建的准确率是100%，即便你的分词器只有90%的准确率，只要顺序正确，它仍然有用。因此，独热向量在神经网络、序列到序列的语言模型和生成语言模型中得到了广泛使用。对于任何需要保留原始文本所有含义的模型或NLP流水线，它都是一个很好的选择。

提示 独热编码器（vectorizer）不会丢弃文本中的任何信息，但我们的分词器会。我们使用的正则表达式分词器会丢弃出现在词语之间的空白字符（ `\s` ），所以无法使用**反分词器（detokenizer）**完美地还原原始文本。而像 spaCy 这样的分词器则会记录这些空白字符，因此可以完美地进行反分词。SpaCy 这个名称就来源于它在精确记录空白字符方面的表现。

这个独热向量序列就像是一种原始文本的数字记录。如果你眯起眼睛看，也许可以把这个由0和1组成的矩阵想象成自动演奏钢琴的纸带，或者是音乐盒金属鼓上的凸点。顶部的词汇表相当于告诉机器每一行应该演奏哪个“音符”或单词，类似于图2.2所示。与自动钢琴或音乐盒不同，你的机械化单词记录器和播放器每次只能使用一个“指头”，因此单词之间的间距不会变化。

Figure 2.2 A player piano roll (Source: CC-BY-SA 4.0: https://commons.wikimedia.org/wiki/File:Piano_Roll_Open.png)

重点在于，你已经将一句自然语言的句子转化成了一系列数字或向量。现在，计算机就能读取这些向量并进行数学运算，就像处理任何其他向量或数字列表一样。这样一来，你就可以将这些向量输入任何需要该向量表示形式的NLP流水线中。从第5章到第10章讨论的深度学习流水线通常都需要这种表示方式，因为它们可以从这些原始的文本表示中提取“特征”以理解其含义，而深度学习流水线可以进一步从数值表示中生成文本。因此，你的自然语言生成（NLG）流水线输出的词流，往往就由一串串独热向量表示，就像自动钢琴为《西部世界》中的观众演奏的一首歌一样。

现在，你所需要做的就是想办法如何构建一个能够**理解**并以新方式组合这些词向量的“自动演奏钢琴”。最终，你可能希望你的聊天机器人或NLP流水线能“演奏”出一首新歌，或者说出你从未听过的话。在第9章和第10章中你将学到如何实现这一点，那时你会学习到循环神经网络（recurrent neural networks），它们在处理独热编码（one-hot encoded）序列方面非常有效。

使用独热词向量对一个句子进行表示，保留了原始句子的所有细节、语法和词序，你已经成功地将单词转化为计算机可以“理解”的数字。这些数字是计算机非常擅长处理的一种特殊数字：二进制数字。但对于一小段句子来说，这就是一个很大的表格。如果你仔细想想，就会意识到这样做实际上是扩大了文档的文件大小。对于较长的文档，这种方式可能并不实际。

那么，这种无损（lossless）的数字化表示到底有多大？你的词汇表大小（即向量的长度）将变得非常庞大。英文中常见单词就有2万个，如果再算上专有名词和其他词汇，总数可能达到数百万。每当你处理一个文档，就需要为它分配一张新的表（矩阵），这就像是你的文档的一张原始图像。如果你做过任何图像处理，就知道必须进行降维，才能从数据中提取有用信息。

我们来算一笔账，以便让你体会这些“钢琴卷纸”到底有多庞大、难以管理。在多数情况下，你在NLP流水线中使用的token词汇表将远远超过10,000或20,000个token——有时甚至可以达到数十万、数百万。假设你的词汇表中有100万个token，你手头上有3000本书，每本书大约3500个句

子、每句15个词（这是短篇读物的合理平均数）。每本书对应一个矩阵，这意味着会产生非常多的大型表格。

这一切将占用约157.5 TB的数据量，这远远超过百万兆字节（MB）；即便你极其高效，每个数只用一个字节存储，你也很难把这些数据存进硬盘。如果每个单元使用一个字节，你需要将近20 TB的存储空间，才能处理一个小型书架上的图书。幸运的是，你实际上并不会使用这种数据结构来长期存储文档。你只会在RAM中临时使用它，在处理文档时每次处理一个词。

因此，存储所有这些0值并记录所有文档中单词的顺序既不实际也不太有用。这种数据结构没有对自然语言文本进行抽象或泛化。一个NLP流水线如果像这样设计，却没有进行任何特征提取或降维，是无法在真实世界中良好运作的。

你真正想做的是将文档的含义压缩为它的“本质”。你希望能将整个文档压缩成一个单一向量，而不是一张大表。你愿意放弃完美的“回忆”（recall），只想抓住文档中大部分的含义（信息），而不是全部。

2.5.2 词袋向量（Bag-of-words vectors）

有没有办法把那些自动演奏钢琴乐谱全部压缩成一个向量？向量是表示任意对象的一种极好方式。借助向量，我们可以仅通过计算欧几里得距离来比较文档之间的相似度。向量让我们能够把线性代数工具用于自然语言处理，而这正是自然语言处理的核心目标：对文本做数学处理。

假设我们可以忽略文本中单词的顺序。首次尝试将文本表示为向量时，我们可以将文本中的词打乱顺序，统统放入一个“袋子”里 —— 对每个句子或短文各放一个袋子。事实证明，单纯知道一篇文档中包含哪些词就能为你的自然语言理解（Natural Language Understanding，NLU）流程提供大量信息。这其实就是大型互联网搜索引擎所使用的表示方式。即使是长度达数页的文档，词袋（Bag-of-words，BOW）向量在总结文档要义时也非常有用。

让我们看看在将《偷书贼》（*The Book Thief*）中的文本词语打乱并计数后会发生什么：

```
>>> bow = sorted(set(re.findall(pattern, text)))
>>> bow[:9]
['!', '.', 'Liesel', 'Trust', 'and', 'arrived', 'clouds', 'hands', 'her']
>>> bow[9:19]
['hold', 'in', 'like', 'me', 'on', 'out', 'rain', 'she', 'the', 'their']
>>> bow[19:27]
['them', 'they', 'though', 'way', 'were', 'when', 'words', 'would']
```

即便是这样的打乱顺序的词袋向量，也可以让你大致看出这个句子是关于 *Trust*（信任）、*words*（词语）、*clouds*（云）、*rain*（雨）以及某个叫 *Liesel*（莉赛尔）的人。你可能注意到的一点

是 Python 的 `sorted()` 会将标点符号排在字母前面，将大写字母排在小写字母前面。这是 ASCII 和 Unicode 字符集中的排序规则；不过，词汇的排序顺序无关紧要。只要你以这种方式对所有文档进行分词，那么机器学习流程在处理任意词汇顺序时都会有一致的表现。

你可以使用这种 BOW 向量方法来压缩文本中的信息内容，将每篇文档转换为更易于处理的数据结构。比如用于关键词搜索时，你可以将文本转换为类似自动演奏钢琴卷轴的独热向量（one-hot vector）并合并成一个二进制的 BOW 向量。在自动演奏钢琴的类比中，这就像同时演奏旋律中的多个音符来组成一个“和弦”。而不是像以前那样在你的 NLU 流程中一次只处理一个词，而是为每篇文档创建一个完整的 BOW 向量。

你可以用这个向量来表示整个文档。由于向量长度必须统一，因此 BOW 向量的长度要等于你的词汇表大小，也就是文档中所有唯一词汇的总数。你可以选择忽略某些不会作为搜索词或关键词出现的词，这些词对你来说可能毫无用处。这就是为什么在进行 BOW 分词时常常忽略停用词（stop words）。对于搜索引擎索引或信息检索系统的初筛阶段来说，这是一种极其高效的表示方式。

这种方法对于帮助机器“理解”单词集合的数学表示至关重要。如果你将词汇限制为最重要的 10,000 个词汇，就可以将一个 3,500 句子的书压缩成约 10KB，或者将 3,000 本书压缩成约 30MB。而如果使用独热向量序列来表示这些文本，即使是这样一个中等规模的语料库也会需要几百 GB 的存储空间。

BOW 表示还有一个优势是，它可以让你在常数时间内（ $O(1)$ ）找到语料库中与目标文本相似的文档。你几乎找不到比这更快的方式了。BOW 向量表示正是实现网页级全文检索引擎高速搜索的关键。在计算机科学和软件工程中，你总是在寻找能实现这种速度的数据结构。所有主流全文检索工具都会使用 BOW 向量来帮助你快速找到目标内容。你可以在 Elasticsearch、Solr、PostgreSQL 中看到自然语言的这种向量化表示，甚至还有一些最先进的搜索引擎，如 Qwant、Searx 和 Wolfram Alpha 也在使用。

幸运的是，你的词汇表中词语在文本中的使用是稀疏的。而对于大多数 BOW 应用，我们通常将文档缩短，有时甚至精简为一句话。因此相比一次弹出所有音符，BOW 向量更像是一种和谐悦耳的钢琴和弦 —— 各个词（音符）配合良好、富有意义。即使文中存在大量不常共现的“失谐”词汇，你的自然语言生成（NLG）流程或聊天机器人依旧可以理解这些和弦。事实上，即使这些“失谐”也是有价值的信息，它们可以帮助机器学习流程理解陈述的语义。

BOW 向量是用于将文档的词标记存储为二进制向量的数据结构，表示某个特定词是否在某个特定句子中出现。这种对一组句子的向量表示可以被“索引化”，以指示哪些词在哪些文档中出现。这个索引相当于你在很多教材结尾看到的索引，但它不是记录哪个词出现在第几页，而是记录在哪句

话（或哪个向量位置）中出现。而传统的图书索引通常只包含与主题有关的词汇，而你则会记录所有词汇的出现位置（至少目前是这样）。

2.5.3 为什么不用字符袋？（Why not bag of characters?）

为什么我们用词袋，而不是字符袋，来表示自然语言文本？对于试图破译未知信息的密码分析人员来说，分析文本中字符的频率是个不错的方法；但对于你本国语言的自然语言文本来说，词语才是更优的表示方式。这个道理很简单：只要你想清楚我们使用这些词袋向量是为了什么。

想一想：你有很多方法可以衡量事物之间的“接近程度”。你大概能很好地判断出谁是你的直系亲属，或你常去的咖啡馆距离你有多近。但你知道如何衡量两段文本之间的“接近”吗？在第4章中你将学习如何用编辑距离（edit distance）来比较两个字符串的相似度，但那其实无法真正捕捉你所关心的文本“含义”。

比如，下面这两句话在你看来有多接近？

I am now coming over to see you.
I am not coming over to see you.

你看出区别了吗？你更愿意收到哪一条作为你朋友发来的邮件？词语 *now* 和 *not* 意思截然不同，尽管拼写几乎一样。这说明：即便只是一个字符的不同，也可能改变整个句子的含义。

如果你只是数了数不同的字符数量，这里是1个，然后除以最长句子的字符数以将距离标准化到0到1之间。此处是 $1/32 = 0.03125$，也就是约3%。为了表示接近度，你可以用 1 减去这个距离，得到 $1 – 0.03125 = 0.96875$，也就是约97%。但你觉得这两句话是97%相似的吗？它们的意义正好相反，我们显然需要更好的度量方式。

如果你比较的是词而不是字符，那么这两个句子中只有一个词不同（7个词中改了1个），这比32个字符中改1个要更合理。它们的接近度会是 $6/7 = 0.8571$，也就是约85%。虽然这个值稍低，但反映得更准确 —— 这正是我们想要的。对于自然语言来说，你不希望“接近度”或“距离度量”仅仅依赖于字符级别的差异计数。这就是为什么在处理自然语言时，我们倾向于用词语作为“有意义的标记”。

再看这两句话：

She and I will come over to your place at 3:00.
At 3:00, she and I will stop by your apartment.

这两句话在含义上接近吗？它们字符长度完全相同，用到了一些相同词语，甚至有些是同义词。但词语和字符的顺序不同，所以我们的表示方式必须能够忽略词序。BOW 向量正是通过为词汇表中的每个词创建一个位置（或槽位）来实现这一点的。

稀疏表示（Sparse representations）

你也许会想：处理超大语料库时，你的词汇量可能会达到成千上万，甚至上百万。这意味着你需要在向量中存储大量的0，比如对于一个包含20个词的句子来说，大部分位置都是0。

对于 *Liesel* 那句话来说，用 `dict` 表示会比用向量节省很多内存；0/1值的映射对内存更友好。但你无法对 `dict` 直接进行数学运算。这就是为什么 `CountVectorizer` 使用稀疏 NumPy 数组来保存词频向量。用字典或稀疏数组来表示向量可以做到：只有当词汇表中的某个词出现在某文档中时才存储1。

如果你想检查单个向量是否正确，最好的方式是使用 pandas 的 `Series` 。你可以把这些向量包装成 pandas 的 `DataFrame` ，以便添加更多句子到你的二值向量语料中。

这种将文档表示为二值向量的方式曾长期被用于文档检索与搜索。所有现代CPU都有内置的内存寻址指令，能够高效地对大规模二值向量集合进行哈希、索引和搜索。虽然这些指令原本是为其他目的设计的（例如内存位置索引以从RAM中检索数据），但它们在文本搜索与检索时处理二值向量也同样高效。我们将在下一章继续探索 BOW 以及它更强大的“近亲”——TF-IDF。

2.6 情感（Sentiment）

无论你在 NLP 流水线中使用的是原始的**标记（token）**、**n-gram 标记序列**、**词干（stem）还是词形还原形式（lemma）**，每一个标记都携带某种信息。其中一个关键的信息就是它所表达的情绪倾向（sentiment）——也就是这个词带来的整体情感或态度。**情感分析（sentiment analysis）**，即判断文本或片段的情绪属性，是自然语言处理中的常见任务，在许多公司中甚至是 NLP 工程师的核心工作。

NLP 流水线能够自动处理用户每天提交的大量反馈内容，节省了企业在客服和分析上的巨大人力成本。企业希望了解用户对其产品的真实看法，因此通常会设置反馈渠道。比如 Amazon 或 Rotten Tomatoes 提供星级评分，这是对情绪倾向的结构化采集方式；但更自然的方式是让用户通过自然语言写评论。提供一个自由文本框，让用户直接表达感受，往往能采集到更有价值的细节。

过去这类评论都需要人工逐条阅读。毕竟，人类才具备理解自然语言中情绪和语气的能力，不是吗？但面对海量评论时，这项任务既繁琐又容易出错。人类处理批量负面反馈时往往效率低下、情绪易受影响，而用户在写反馈时也不会刻意保持理性与客观。

相比之下，机器既不会被触发情绪，也不带有偏见。更重要的是，处理自然语言并提取有用信息甚至意义，并不是人类独有的能力。一个 NLP 流水线可以快速、系统地处理成千上万条文本评论，客观地输出数值型结果，指示每条文本的情绪是正面、负面，还是中性或复杂。

另一个典型应用是识别和屏蔽垃圾评论或网络喷子。在这类任务中，你希望聊天机器人能够分析接收到的信息的情绪倾向，并据此做出合适的响应。进一步来说，你还希望它在准备说话前也能分析一下即将发出的语句是否“友好”。正如我们常听到的一句话：“如果你说不出什么好话，那就别说。”为此，你需要让机器人在发言前评估语句的情绪强度，并决定是否继续发出。

你要如何设计一个流程来衡量一段文本的情绪倾向程度？例如你只关心积极或喜欢的程度——用户对某产品或服务的好感度。你可以让 NLP 流水线输出一个介于 –1 到 +1 之间的浮点值：+1 表示强烈的正面情绪，–1 表示强烈的负面情绪。接近 0 的值（如 +0.1 或 –0.1）代表中性或模棱两可的态度，例如“还行，有好有坏”。

情感分析的实现方法主要有两种：

- 基于规则的分析；
- 基于机器学习的分析。

第一种方法依赖人类设定的规则（通常是启发式规则）来判断情绪倾向。最常见的做法是查找文本中出现的关键词，并将这些关键词映射到一个字典（如 Python 的 `dict`）中，赋予它们情绪得分。既然你已经了解如何进行**标记化处理**，那么你可以用词典来映射**词干（stem）**、**词形还原形式（lemma）或n-gram 标记序列**。基本的处理逻辑是：找到所有命中词典的标记，把它们的分数累加起来。这当然要求你提前编写好这个情绪词典。我们将在下一节展示如何使用 `scikit-learn` 中的 VADER 工具来完成这一任务。

第二种方法依赖机器学习，通过大量已标注情绪倾向的文本训练出模型，自动推理判断规则。这种模型的输入是一段文本，输出是一个表示情绪倾向程度的数值（如正向概率、垃圾程度或攻击性指数）。要使用机器学习方法，你需要大量已标注的训练数据。Twitter 是一个常见数据源，因为标签词（如 #awesome、#happy、#sarcasm）能被用来自动创建“自标注”语料库。如果你所在的公司拥有产品评论和相应的星级评分，那么这些评分就可以作为标签，用来训练情绪倾向的预测模型。在你完成 VADER 分析之后，我们将介绍如何处理这类数据，并使用**朴素贝叶斯（naive Bayes）**算法构建一个基于标记输入的模型。

2.6.1 VADER：基于规则的情感分析器（A rule-based sentiment analyzer）

乔治亚理工学院（Georgia Institute of Technology）的研究人员 Eric Gilbert 和 CJ Hutto 创建了首批成功的基于规则的情感分析算法之一。他们将这一算法命名为 **Valence Aware Dictionary and Sentiment Reasoner（VADER）**。许多 NLP 工具包都实现了该算法的某种变体，例如 NLTK 就在其 `nltk.sentiment.vader` 模块中集成了 VADER。Hutto 曾亲自维护 VADER

的 Python 包 `vaderSentiment`，即便他在 2020 年停止维护，该工具仍广受欢迎。本节中我们将直接使用该开源工具 `vaderSentiment` 。

你需要先运行以下命令安装它：

```
pip install vaderSentiment
```

接下来是示例代码：

```
>>> from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
>>> sa = SentimentIntensityAnalyzer()
>>> sa.lexicon
{
  ':(': -1.9,
  ':)': 2.0,
  ...
  'pls': 0.3,
  'plz': 0.3,
  ...
  'great': 3.1,
  ...
}
>>> [(tok, score) for tok, score in sa.lexicon.items() if " " in tok]
[("(' !')", 1.6), ("can't stand", -2.0), ('fed up', -1.8), ('screwed up', -1.5)]
>>> sa.polarity_scores(text="Python is very readable and it's great for NLP.")
{'compound': 0.6249, 'neg': 0.0, 'neu': 0.661, 'pos': 0.339}
>>> sa.polarity_scores(text="Python is not a bad choice for most applications.")
{'compound': 0.431, 'neg': 0.0, 'neu': 0.711, 'pos': 0.289}
```

🔹 `SentimentIntensityAnalyzer.lexicon` 包含了一个**情绪词典**，其中记录了标记与其对应的情感得分。

🔹 标记化器（tokenizer）必须能识别标点和表情符号（例如 emoji），因为它们常用于表达情感。

🔹 如果你的 NLP 流水线中包含词干提取或词形还原处理，需要确保将处理后的标记映射回 VADER 的词典。

🔹 VADER 总共定义了大约 7500 个标记，其中仅有极少数是包含空格的 n-gram 标记。

🔹 它将每段文本分解为正向、负向和中性情绪三个部分，并最终合成为一个复合情绪得分（compound score）。

> 🔷 它对否定词处理得很好（例如：great 比 not bad 得分更高），但不考虑词典外的词，也不处理 n-gram。

然后我们使用它来分析前面提到的示例句子：

```
>>> corpus = ["Absolutely perfect! Love it! :-) :-) :-)",
...           "Horrible! Completely useless. :(",
...           "It was OK. Some good and some bad things."]
>>> for doc in corpus:
...     scores = sa.polarity_scores(doc)
...     print('{:+.4f}: {}'.format(scores['compound'], doc))
```

输出结果如下：

```
+0.9428: Absolutely perfect! Love it! :-) :-) :-)
-0.8768: Horrible! Completely useless. :(
-0.1531: It was OK. Some good and some bad things.
```

这看起来非常接近我们的预期：VADER 能给出直观合理的情感得分。然而它也有一个局限：**它不会处理文档中的所有词汇**。VADER 只“认识”大约 7500 个硬编码在算法中的词汇。如果你希望更多词汇也能参与得分，或者你不想自己为成千上万的词汇构建情绪得分词典，并添加到 `SentimentIntensityAnalyzer.lexicon` 中，那这个基于规则的方法就显得局限了。

对于你不熟悉的语言，基于规则的方法通常也难以适用，因为你并不知道该如何为词典中的词分配合适的情感得分。这正是机器学习情感分析器存在的意义。

2.6.2 朴素贝叶斯（Naive Bayes）

朴素贝叶斯（*naive Bayes*）模型会尝试在一组文档中找出能够预测目标（输出）变量的关键词。当目标变量是情绪倾向时，模型会识别出那些能够预测该情绪的标记（token）。朴素贝叶斯的优势之一在于，其内部的系数会将标记映射为得分，就像 VADER 那样。不过这一次，不再是人手工指定这些得分，而是由机器自动学习出最“优”的得分。

对于任意机器学习算法，你首先需要准备一个数据集。你需要的是若干带有正向情感标签的文本（正面情绪得分）。在构建 VADER 时，Hutto 就整理了四个情感数据集。你将使用其中一个，并通过 `nlpia2` 包加载它们：

```
>>> movies = pd.read_csv('https://proai.org/movie-reviews.csv.gz', index_col=0)
>>> movies.head().round(2)
```

数据前几行如下：

```
id    sentiment    text
1     2.27         The Rock is destined to be the 21st Century's ...
2     3.53         The gorgeously elaborate continuation of 'The...
3    -0.60         Ineffective but too tepid biopic
4     1.47         If you sometimes like to go to the movies to h...
5     1.73         Emerges as something rare, an issue movie that...
```

```
>>> movies.describe().round(2)
```

Plain Text

```
       sentiment
count   10605.00
mean        0.00
std         1.92
min        -3.88
...
max         3.94
```

> 可以看到：
>
> - 情感得分（即电影评分）经过居中处理（mean 为 0）；
> - 分布范围大约是 -4（最差）到 +4（最佳）。

看起来这些电影评论确实是经过**中心化的** —— 即通过减去平均值来将所有情感评分标准化，使平均值为 0，避免偏向任一方向。并且可以推测，情感评分的有效范围是 -4 到 +4。

接下来，你可以将这些影评文本进行标记化，并为每一条创建一个 BOW（词袋向量）。将它们全部装入 pandas 的 `DataFrame` 中将更便于操作：

```
>>> import pandas as pd
>>> pd.options.display.width = 75
>>> from nltk.tokenize import casual_tokenize
>>> bows = []
>>> from collections import Counter
>>> for text in movies.text:
...     bows.append(Counter(casual_tokenize(text)))
>>> df_movies = pd.DataFrame.from_records(bows)
>>> df_movies = df_movies.fillna(0).astype(int)
>>> df_movies.shape
(10605, 20756)
```

- `casual_tokenize` 能比 `TreebankWordTokenizer` 更好地处理表情符号、非常规标点和俚语；
- `Counter` 会统计每条文本中的词频，结果是一个以标记为键、出现次数为值的字典；
- `from_records()` 会将字典列表转换为表格，每个标记对应一列，空值会变成 NaN；
- 使用 `.fillna(0).astype(int)` 将 NaN 转换为 0，得到稀疏整数矩阵。

```
>>> df_movies.head()
   !  "  ‚  $  %  &  ' ...  zone  zoning  zzzzzzzzz  ½  élan  -  ’
0  0  0  0  0  0  0  4 ...     0       0          0  0     0  0  0
1  0  0  0  0  0  0  4 ...     0       0          0  0     0  0  0
2  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0
3  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0
4  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0

>>> df_movies.head()[list(bows[0].keys())]
   The  Rock  is  destined  to  be ...  Van  Damme  or  Steven  Segal  .
0    1     1   1         1   2   1 ...    1      1   1       1      1  1
1    2     0   1         0   0   0 ...    0      0   0       0      0  4
2    0     0   0         0   0   0 ...    0      0   0       0      0  0
3    0     0   1         0   4   0 ...    0      0   0       0      0  1
4    0     0   0         0   0   0 ...    0      0   0       0      0  1

[5 rows x 33 columns]
```

当你没有使用大小写标准化（case normalization）、停用词过滤器（stop word filters）、词干提取（stemming）或词形还原（lemmatization）时，你的词汇表会变得非常庞大，因为你必须记录每一个拼写差异或大小写的细微变化。你可以尝试在你的 NLP 流水线中插入一些降维（dimension reduction）步骤，以观察这些处理对流水线准确率和存储所有这些词袋向量（BOW）所需内存的影响。现在，我们已经拥有了训练情感预测器所需的全部数据，并可以在训练集上计算损失函数。

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> nb = MultinomialNB()
# Naive Bayes 模型是分类器，因此你需要将输出变量（情感浮点数）转换为
# 离散标签（整数、字符串或布尔值）。
>>> nb = nb.fit(df_movies, movies.sentiment > 0)

# 将你的离散分类变量转换为一个介于 -4 和 +4 之间的真实值，
# 以便你可以将其与“真实”情感进行比较
>>> movies['pred_senti'] = (
...     nb.predict_proba(df_movies))[:, 1] * 8 - 4
>>> movies['error'] = movies.pred_senti - movies.sentiment

# 平均预测误差的绝对值，或平均绝对误差（MAE）
>>> mae = movies['error'].abs().mean().round(1)
>>> mae
1.9
```

为了创建一个二元分类标签，你可以利用一个事实：当评论的情感为正时，居中的电影评分（情感标签）为正（大于零）：

```
>>> movies['senti_ispos'] = (movies['sentiment'] > 0).astype(int)
>>> movies['pred_ispos'] = (movies['pred_senti'] > 0).astype(int)
>>> columns = [c for c in movies.columns if 'senti' in c or 'pred' in c]
>>> movies[columns].head(8)
```

```
    sentiment  pred_senti  senti_ispos  pred_ispos
id
1    2.266667    2.511515            1           1
2    3.533333    3.999904            1           1
3   -0.600000   -3.655976            0           0
4    1.466667    1.940954            1           1
5    1.733333    3.910373            1           1
6    2.533333    3.995138            1           1
7    2.466667    3.960466            1           1
8    1.266667   -1.918701            1           0
```

```
>>> (movies.pred_ispos ==
...  movies.senti_ispos).sum() / len(movies)
0.9344648750589345 # 你有 93% 的时间正确预测了“点赞”评分
```

这是构建情感分析器的一个相当不错的开始，只用了几行代码（和大量数据）。你不需要猜测某个 7500 词的词表中各个词的情感，并将它们硬编码到像 VADER 这样的算法中。相反，你告诉机器整个文本片段的情感评分，然后机器完成了剩下的工作，找出了每个词的情感。这就是机器学习和自然语言处理（NLP）的力量！

你认为这个模型在迁移到完全不同的文本类型（比如产品评论）时表现如何？人们在描述自己喜欢的电影和产品（如电子产品和家用商品）时是否会使用相同的词？很可能不会——但通过从不同领域获取具有挑战性的文本来测试你的语言模型的健壮性是一个好主意。通过在新领域上测试你的模型，你可以获得更多示例和数据集的思路，用于训练集和测试集。

首先，你需要加载产品评论。看一下你加载的文件内容，以确保你了解数据集中包含了什么：

```
>>> products = pd.read_csv('https://proai.org/product-reviews.csv.gz')
>>> products.columns
Index(['id', 'sentiment', 'text'], dtype='object')
>>> products.head()
```

```
   id  sentiment                                               text
0  1_1      -0.90    troubleshooting ad-2500 and ad-2600 no picture...
1  1_2      -0.15    repost from january 13, 2004 with a better fit...
2  1_3      -0.20    does your apex dvd player only play dvd audio ...
3  1_4      -0.10    or does it play audio and video but scrolling ...
4  1_5      -0.50    before you try to return the player or waste h...
```

接下来，你需要加载产品评论：

```
>>> bows = []
>>> for text in products['text']:
...     bows.append(Counter(casual_tokenize(text)))
>>> df_products = pd.DataFrame.from_records(bows)
>>> df_products = df_products.fillna(0).astype(int)
>>> df_products.shape # 产品评论的 BOW（词袋模型）有一个与电影评论不同的词汇表
```

当你将一个 BOW 向量的 DataFrame 与另一个合并时会发生什么？

```
>>> df_all_bows = pd.concat([df_movies, df_products])
>>> df_all_bows.columns # BOW 向量的列包含标记字符串
Index(['!', '"',
...
 'zoomed', 'zooming', 'zooms', 'zx', 'zzzzzzzzz', ...],
dtype='object', length=23302)
```

合并后的 BOWs 的 DataFrame 拥有在产品评论中出现但不在电影评论中的词汇。现在你有 23,302 个用于电影评论和产品评论的唯一词项。电影评论中只包含 20,756 个唯一词项，因此必然有 23,302 − 20,756，即 2,546 个关于产品的新词项此前未在你的词汇表中。

为了使用你的 Naive Bayes 模型对产品评论进行预测，你需要确保新的产品 BOWs 在列（词项）顺序上与用于训练模型的原始电影评论完全一致。毕竟，该模型没有与这些新词项的交互经验，因此不知道哪些词项适合哪些权重，你也不希望它搞错产品评论中的词项：

```
>>> vocab = list(df_movies.columns) # 电影评论词汇表
>>> df_products = df_all_bows.iloc[len(movies):] # 移除电影评论
>>> df_products = df_products[vocab] # 从你的词汇表中移除所有新的产品评论词项
>>> df_products = df_products.fillna(0).astype(int)
>>> df_products.shape
(3546, 20756)
>>> df_movies.shape # 电影的 BOW 向量与产品评论具有相同大小的词汇表（列）
(10605, 20756)
```

现在，你的两组向量（DataFrames）都具有 20,756 列或唯一词项。接下来，你需要将产品评论的标签转换成二元情感标签，以模拟你为原始 Naive Bayes 模型训练的电影评论分类标签：

```
>>> products['senti_ispos'] = (products['sentiment'] > 0).astype(int)
>>> products['pred_ispos'] = nb.predict(df_products).astype(int)
>>> correct = (products['pred_ispos']
...            -= products['senti_ispos']) # 当预测的情感与数据集中的标签一致时，即为正确预测
>>> correct.sum() / len(products)
0.557...
```

因此你的 Naive Bayes 模型在预测产品评论情感是正面（点赞）或负面（点踩）时表现不佳，只比抛硬币略好一点。这种表现不佳的原因之一是你的产品评论词汇表来自 `casual_tokenize` 的文本中，有 2,546 个词项未在电影评论中出现。这大约占原始电影评论词汇表的 10%，意味着所有这些词在你的 Naive Bayes 模型中都不会有任何权重或得分。此外，Naive Bayes 模型对否定词的处理不如 VADER。你需要在分词器中引入 n-gram，以将否定词（如 not 或 never）与它们所要修饰的正面词连接起来。

我们把继续改进这个机器学习模型的 NLP 动作留给你。你可以在每一步都与 VADER 的进展进行比较，以确定机器学习是否是比硬编码算法更好的 NLP 方法。

2.7 自我测试

1. 相较于词级分词器，WordPiece 分词器（例如 BPE）有哪些优点?
2. 词形还原器（lemmatizer）与词干提取器（stemmer）之间有何区别？通常情况下哪一个更好?
3. 词形还原器是如何提高搜索引擎（例如 You.com）返回包含你所查找内容的搜索结果的可能性的?
4. 大小写归一化、词形还原或停用词去除会提升你常规 NLP 流水线的准确性吗？对于像识别误导性新闻标题（标题党）这类问题又会如何?
5. 你的标记计数中是否有统计信息可以帮助你决定在 n-gram NLP 流水线中使用的 n 值?
6. 是否有网站可以下载大多数词和 n-gram 的词频数据?

本章小结

- 你已经实现了分词并为你的应用配置了一个分词器。
- n-gram 分词有助于保留文档中的部分“词序”信息。
- 归一化和词干提取会将词汇合并为一组，从而提升搜索引擎的“召回率”，但会降低准确率。词形还原和像 `casual_tokenize()` 这样的自定义分词器可以提高准确率并减少信息丢失。
- 停用词可能包含有用信息，丢弃它们并不总是有益的。

本章注释

1. Lysandre 在 Hugging Face 文档中解释了各种分词器选项（https://huggingface.co/transformers/tokenizer_summary.html）。
2. Markus Zusak，《偷书贼》，第85页。
3. Peter Watts，《盲视》。
4. 感谢 Wiktor Stribiżew（https://stackoverflow.com/a/43094210/623735）。
5. 第 2 章的 `nlpia2` 源代码（https://proai.org/nlpia2-ch2）包含额外的 spaCy 和 displaCy 选项及示例。

6. “Python NLP 分词器的基准测试”，Andrew Long（https://towardsdatascience.com/benchmarking-python-nlp-tokenizers-3ac4735100c5）。
7. “Grapheme,” 维基百科（https://en.wikipedia.org/wiki/Grapheme）。
8. Suzi Park 和 Hyopil Shin，《形态丰富语言中词嵌入的字素级感知》（https://www.aclweb.org/anthology/L18-1471.pdf）。
9. 如果你想更深入了解“词”的定义，请阅读 Jerome Packard 所著《汉语的形态学》导论部分，其中详细讨论了“词”这一概念。事实上，在 20 世纪以前，这一概念在中文中几乎完全不存在，它是从英语语法中翻译引入的。
10. 在许多应用中，术语“n-gram”更常用于指字符 n-gram，而不是词 n-gram。例如，主流关系型数据库 PostgreSQL 使用三元索引，它会将文本划分为字符三元组，而不是词三元组。在本书中，我们使用“n-gram”来指代词语序列，“character n-gram”来指代字符序列
11. Hannes 和 Cole 可能正在尖叫：“我们早就说过了！”
12. 实际上，用于 BPE 和 WordPiece 分词器的标记字符串表示法会在标记开始或结尾加上标记字符，以表明词边界的缺失（通常是空格或标点）。因此你可能会在 BPE 词汇中看到 `aphr##` 这样的标记，用来表示 aphrodisiac 中的词根 aphp（https://stackoverflow.com/a/55416944/623735）。
13. 美国最近通过了歧视性的选民限制法案。“威斯康星州新冠疫情下的选民注册限制影响黑人居民”，Claire Campbell 和 Laura Schultz（https://apnews.com/article/health-wisconsin-voting-voter-registration-2019-2020-coronavirus-pandemic-2cb40feb988eea16a184fe11db6d81a4）。
14. 参见第 7 章关于 BPE 用于翻译的内容。
15. Lysandre 在 Hugging Face 的 transformers 文档中解释了子词分词器的各种变体（https://huggingface.co/transformers/tokenizer_summary.html）。
16. 参见 Hugging Face 的分词器文档（https://huggingface.co/docs/transformers/tokenizer_summary）。
17. 语言学和 NLP 技术常用于从 DNA 和 RNA 中提取信息。以下网站提供氨基酸符号表，可帮助你将氨基酸语言翻译成人类可读语言：“Amino Acid,” Wikipedia（https://en.wikipedia.org/wiki/Amino_acid#Table_of_standard_amino_acid_abbreviations_and_properties）。
18. 你可能在数据库课程或 PostgreSQL（`postgres`）文档中学到过三元索引，但这些是字符三元组。它们帮助你在包含海量字符串的数据库中使用 `%` 和 `~*` SQL 全文检索语法快速

模糊匹配字符串。

19. NLTK 的语料库中提供了多种语言的停用词列表（https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip）。
20. 参见“文本数据和自然语言处理分析”（http://rstudio-pubs-static.s3.amazonaws.com/41251_4c55df8747c4850a7fb26fb9a969c8f.html）。
21. spaCy 包含一个可自定义的停用词列表，可通过以下 Stack Overflow 回答进行自定义（https://stackoverflow.com/a/51627002/623735）。
22. 如果你想帮助他人更好地使用 Searx，可以养成写作或搜索时使用“Searx”（发音为 see Rch 的习惯），你可以改变词义，让世界变得更美好！
23. NLTK 包（https://pypi.org/project/nltk）包含了你在很多在线教程中会见到的停用词列表。
24. 在市场营销和 SEO 中，Damien Doyle 维护着一个按搜索引擎排名的站点示例（https://www.ranks.nl/stopwords）。
25. Vadi Kumar 维护了一些停用词列表（https://github.com/Vadi/stop-words）。
26. Ted Chiang，《呼吸》之“Truth of Fact, Truth of Fiction”章节。
27. 参见“Camel Case”，维基百科（https://en.wikipedia.org/wiki/Camel_case_case）。
28. 此处假设 Python 3 中 `str.lower()` 的行为。在 Python 2 中，字节串（字符串）通过简单地将 ASCII 编号范围内的所有字母字符转换为小写来处理；但在 Python 3 中，`str.lower` 能正确转换字符，使其能处理英文中的修饰字符（例如 résumé 中 e 上的变音符）以及非英语语言中的大小写规则。
29. “cup of joe”的三元组（trigram）索引（https://en.wiktionary.org/wiki/cup_of_joe）是 cup of coffee 的俚语。
30. 参见附录 D 了解更多关于精确率和召回率的内容。这里有一个对多个搜索引擎召回率的比较（http://www.webology.org/2005/v2n2/a12.html）。
31. 如果你忘记了如何计算召回率，请回顾附录 D 或访问维基百科页面了解更多（https://en.wikipedia.org/wiki/Precision_and_recall）。
32. 参见 M. F. Porter 所著“An Algorithm for Suffix Stripping”（http://www.cs.toronto.edu/~frank/csc2501/Readings/R2_Porter/Porter-1980.pdf）
33. 参见 M. F. Porter 所著“Snowball: A Language for Stemming Algorithms”（http://snowball.tartarus.org/texts/introduction.html）。

34. 感谢 Kyle Gorman 指出这一点。
35. 额外的元数据也用于调整搜索结果的排序。Duck Duck Go 和其他搜索引擎（如 MetaGer.org、Brave.com、Mwmb1.org 和 Phind.com）结合了数百种算法来对搜索结果进行排序（https://duck.co/help/results/sources）。
36. “Nice guys finish first!”来自《超级合作者》作者 M. A. Nowak。
37. 参见 Stanford 的《信息检索导论》，作者包括 Christopher D. Manning、Prabhakar Raghavan 和 Hinrich Schütze，“Stemming and Lemmatization”一节（https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html）。
38. 可在维基百科文章“Chinese Character Classification”中了解更多（https://en.wikipedia.org/wiki/Chinese_character_classification）。
39. 参见维基百科上的“Player Piano”条目（https://en.wikipedia.org/wiki/Player_piano）
40. 参见维基百科上的“Music Box”条目（https://en.wikipedia.org/wiki/Music_box）。
41. 《西部世界》是一部关于邪恶人类与人形机器人（包括在主吧台演奏钢琴的机器人）之间故事的电视剧。
42. 参见 Apache Solr 主页和 Java 源码（https://solr.apache.org/）。
43. Qwant 是一个基于欧洲的搜索引擎（https://www.qwant.com/）。
44. 参见 Searx 的 GitHub 仓库（https://github.com/searx/searx）及其搜索页面（https://searx.space/）。
45. 参见 Wolfram Alpha 主页（https://www.wolframalpha.com/）。
46. 参见 Hutto 和 Gilbert 所著的《VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text》（https://ojs.aaai.org/index.php/ICWSM/article/download/14550/14399）。
47. 你可以在 GitHub 上找到包含安装说明的 VADER 包源码（https://github.com/cjhutto/vaderSentiment）。
48. 如果你尚未安装 nlpia，请查看以下安装说明：http://gitlab.com/tangibleai/nlpia2。
49. 如果你需要回顾 training set 和 error 等术语，请查阅附录 D，其中包含多个基本机器学习术语的介绍。
50. 提示：当你不确定时，就做实验。这称为超参数调优（hyperparameter tuning）。这里有一个假新闻数据集供你实验使用：https://www.kaggle.com/clmentbisaillon/fake-and-real-news-dataset/download。

51. 提示：一个曾经号称“不作恶”的公司创建了这个庞大的 NLP 语料库。