04.《自然语言处理实战·第2版》第4章 寻找词频背后的含义：语义分析

本章小结

本章注释

本章内容

- 分析语义（meaning）以创建主题向量（topic vectors）
- 利用主题向量之间的语义相似度进行语义搜索（semantic search）
- 针对大型语料进行可扩展的语义分析与语义搜索
- 在 NLP 流水线中将语义成分（主题）作为特征
- 在高维向量空间中导航

通过前几章，你已经学会了不少自然语言处理技巧，但现在也许是第一次可以做一点“魔法”了：机器第一次能够理解单词的**含义（meaning）**。

你在第 3 章学到的 **TF–IDF（term frequency–inverse document frequency，词频–逆文档频率）** 向量帮助你估计了一段文本中单词的重要性。你利用 TF–IDF 向量和矩阵来衡量各单词对整段文本总体含义的重要程度。这些 TF–IDF “重要性”得分不仅适用于单词，也适用于短语或 *n*–gram。

研究者发现了若干方法，用单词与其他单词的共现（co-occurrence）来表示其含义。你将学习其中一些方法，如 **潜在语义分析（latent semantic analysis, LSA）** 和 **潜在狄利克雷分配（latent Dirichlet allocation, LDiA）**。这些方法创建 **语义向量（semantic vectors）** 或 **主题向量（topic vectors）** 来表示单词和文档。你将使用上一章得到的加权词频（TF–IDF 向量或 BOW 向量）及其相关性来计算构成主题向量维度的主题“得分”。

主题向量能完成许多有趣的事情：

- 它们使你能够按**含义**而非关键词来搜索文档——**语义搜索**。绝大多数情况下，语义搜索的结果远优于关键词搜索。有时即使用户想不起准确关键词，语义搜索仍能返回他们需要的文档。
- 语义向量还能帮助识别最能代表陈述、文档或语料主题的单词和 *n*–gram。结合单词及其相对重要性，你可以为某人提供文档最有意义的关键词集合，对其含义进行总结。
- 最终，你将能够比较两句话或两篇文档并判断它们在**含义**上有多“接近”。

TIP *topic*、*semantic* 与 *meaning* 这些词在 NLP 语境中含义相近，常可互换使用。本章你将学习如何构建一个能够自行识别同义关系的 NLP 流水线——比如让模型自行发现短语 *figure it out* 与单词 *compute* 在含义上的关联。机器只能**计算**含义，而非自行“领悟”含义。

你将很快发现，由单词线性组合而成、构成主题向量维度的这些成分，是表达含义的强大表示方式。

4.1 从词频到主题得分

你已经知道如何统计单词频次并在 TF—IDF 向量或矩阵中给出重要性得分。但这还不够。下面看看仅靠词频表示会遇到哪些问题，以及如何着手用**含义**而非单独的词频来表示文本。

4.1.1 TF—IDF 向量与词形还原（lemmatization）的局限

TF—IDF 向量根据单词在文本中的**精确拼写**来计数。若两句话用不同拼写重述相同含义，则可能得到完全不同的 TF—IDF 表示，这会干扰搜索引擎与基于词频的相似度比较。

在第 2 章中，你对词尾进行了规范化，使仅在最后几个字符上有所差异的单词被归并为一个词元；你使用词干提取（stemming）和词形还原（lemmatization）将相似拼写、往往含义相近的单词聚为小集合。接着，你用它们的词干或词形作为新词元进行处理。

然而，这种词形还原方法虽能把拼写**相似**的单词聚在一起，却不一定能把含义相似的单词聚在一起，更无法把同义词配对（比如 *beautiful* 与 *pretty* 拼写毫不相似）。更糟的是，词干提取或词形还原有时会把含义相反的反义词（如 *useful* 与 *useless*）错误地归为一类。

结果是：两段讲述同一主题但用词不同的文本，在你的词形还原 TF—IDF 向量空间中可能距离很远；相反，有时两段含义相距甚远的文本却可能很近。即便使用第 3 章提到的最先进 TF—IDF 相似度分数（Okapi BM25 或余弦相似度），也无法连接这些同义词或区分这些反义词。

例如，本章的 TF—IDF 向量未必与大学教材中关于 *latent semantic indexing*（潜在语义索引）的同义内容接近，因为本章使用现代、口语化的术语；而教授和研究者在教材、讲义里使用的是更规范、严格的语言，而且十年前流行的术语如今也已迅速演变（如 *latent semantic indexing* 当年比 *latent semantic analysis* 更常见）。

不同词但含义相近会给 TF—IDF 带来麻烦；拼写相似但含义大相径庭也同样麻烦。英语单词通常具有多重含义，这对任何学习者都是挑战，也包括机器学习模型：这种“一词多义”现象称为 **polysemy（多义性）**。

多义性会这样影响语义：

- **Homonyms（同音异义词）**——拼写和读音相同但含义不同，例如 (a) *The band was playing old Beatles songs.*（乐队）与 (b) *Her hair band was very beautiful.*（头箍）。
- **Homographs（同形异义词）**——拼写相同但读音或含义不同，例如 (a) *I object to this decision.*（动词，反对）与 (b) *I don't recognize this object.*（名词，物体）。
- **Zeugma（轭辞）**——在同一句话中同时使用一个单词的两个含义，例如 *Mr. Pickwick took his hat and his leave.*

所有这些现象都会降低 TF–IDF 的效果，使得"用相似单词却含义不同"的句子 TF–IDF 向量过于接近。为应对这些挑战，我们需要更强大的工具。

4.1.2 主题向量（Topic vectors）

当你对 TF–IDF 向量做加、减等运算时，这些和与差只能告诉你被合并或相减的文档里单词出现的频率；它们并不能揭示这些词背后的"含义"。你确实可以把 TF–IDF 矩阵自乘，计算词–词 TF–IDF 向量（词共现或相关性向量），但用这些稀疏且高维的向量做向量推理效果并不好；把它们相加或相减并不能很好地表示某个现有概念、单词或主题。

因此，你需要一种方法，从词频统计中提取额外的信息——也就是**含义**。你需要更可靠地估计文档中单词所表达的意义，搞清楚这些单词组合在特定文档里的**意思**。理想情况下，你想用一种类似 TF–IDF 向量、但更紧凑、更有意义的向量来表示这种含义。

本质上，在创建这些新向量时，你会定义一个**新空间**。用 TF–IDF 或词袋（BOW）向量表示单词和文档时，你在一个由文档中出现的单词或词元定义的空间里工作，每个词元占据一维——这很容易就达到几千维。而且每个词元与其他所有词元都是正交的：把代表某个单词的向量与代表另一个单词的向量相乘，总是得到 0，即便这两个单词是同义词。

主题建模（topic modeling） 的过程就是找到一个维度更少的新空间，使得语义上接近的单词在这些维度上对齐。我们称这些维度为 **topics（主题）**，而新空间中的向量称为 **topic vectors（主题向量）**。主题数可按需求任意设置。

本章中你得到的主题向量依旧可以像普通向量那样做加减，但这一次，它们的和与差比在 TF–IDF 空间中意义大得多。主题向量间的距离或相似度对于查找主题相似的文档，或进行语义搜索非常有用（本章末尾会讨论语义搜索）。

把向量转换到新空间后，你的语料中每个文档都有一个 **文档–主题向量**；词汇表中的每个单词也会有一个 **词–主题向量**。因此，要为任何新文档计算主题向量，只需把该文档所有词–主题向量加起来即可。

为单词或句子的语义构建数值表示并不容易，尤其是对英语这种"模糊"的语言——它有多种方言，同一个单词往往存在多种解释。

考虑到存在这些挑战，你能把一个含有一百万维（词元）的 TF—IDF 向量压缩成仅 10 或 100 维（主题）的向量吗？这就像要调出正确的油漆原色配比，好把家里墙上的钉孔补上。

你得找到那些在语义上“属于同一主题”的词维度，把它们各自的 TF—IDF 值加权合并，生成一个新数值来表示该主题在文档中的占比。你甚至可以按每个单词对主题的重要程度来设置权重；对降低文本与该主题相关性的词，还可以赋予负权重。

4.1.3 思想实验（Thought experiment）

下面通过一个思想实验来走一遍流程。假设你有某个文档的 TF—IDF 向量，想把它转换成主题向量，需要考虑每个单词对各主题的贡献度。

设想要处理一些关于纽约中央公园宠物的句子。我们创建三个主题：**petness**（宠物性）、**animalness**（动物性）和 **cityness**（城市性）。例如，*petness* 会对 *cat*、*dog* 之类的词赋高分，但大概率忽视 *NYC*、*apple*。如果不借助计算机，只凭常识“训练”主题模型，可能会得到如下权重（见代码清单 4.1）。

代码清单 4.1 你的主题示例权重 Python

```
>>> import numpy as np

>>> topic = {}
>>> tfidf = dict(list(zip('cat dog apple lion NYC love'.split(),
...                       np.random.rand(6))))   # 随机生成示例 tfidf 值

>>> topic['petness'] = (.3 * tfidf['cat'] +      # 手工设定权重 (3, 3, 0,
0, -2, 2)
...                     .3 * tfidf['dog'] +      # 与虚构的 tfidf 值相乘,
...                     0 * tfidf['apple'] +     # 用于你的假想文档
...                     0 * tfidf['lion'] -
...                     .2 * tfidf['NYC'] +
...                     .2 * tfidf['love'])

>>> topic['animalness'] = (.1 * tfidf['cat'] +
...                        .1 * tfidf['dog'] -
...                        .1 * tfidf['apple'] +
...                        .5 * tfidf['lion'] +
...                        .1 * tfidf['NYC'] -
...                        .1 * tfidf['love'])

>>> topic['cityness'] = ( 0 * tfidf['cat'] -
...                       .1 * tfidf['dog'] +
...                       .2 * tfidf['apple'] -
...                       .1 * tfidf['lion'] +
...                       .5 * tfidf['NYC'] +
...                       .1 * tfidf['love'])
```

在此**思想实验（thought experiment）**中，我们把可能指示各主题的词频相加，并按照单词与主题相关联的可能性（TF–IDF 值）为其加权。这些权重也可以为负，用于表示与某主题在某种意义上“相反”的单词。

这并不是真正的算法或示例实现，只是一个思想实验：你只是想弄清楚如何教机器像你一样思考。你任意决定仅将单词及文档分解为三个主题（**petness**、**animalness**、**cityness**），而且你的词汇量受限——仅包含六个单词。

下一步是思考：人类如何用数学方式决定哪些主题与单词相连，以及这些连接应有多大权重。一旦确定要建模的三个主题，你必须为这些主题中每个单词的权重做决策。你按各主题的比例混合单词，以制作主题的“调色配方”。主题建模转换（配色配方）可表示为一个 3 × 6 的比例（权重）矩阵，将三个主题连接到六个单词。你把这个矩阵与一个假想的 6 × 1 TF–IDF 向量相乘，就得到该文档的 3 × 1 主题向量。

你主观判断 *cat* 与 *dog* 对 **petness** 主题贡献相当（权重为 0.3）。因此，在 TF–IDF→主题转换矩阵的左上角，那两个值都是 0.3。你能想象用软件如何计算这些比例吗？记住，你手头有大量文档可供计算机读取、分词并统计词频，你可对想要的任意数量文档计算 TF–IDF 向量。阅读后续内容时，继续思考如何利用这些计数来为单词计算主题权重。

你决定 *NYC* 对 **petness** 主题应有负权重。某种意义上，城市名（以及一般的专有名、缩写、首字母缩写）与宠物单词共享的含义很少。想想单词之间“共享含义”意味着什么——在 TF–IDF 矩阵中是否存在某种可代表单词共享含义的东西?

注意 *apple* 在 **cityness** 主题向量中具有很小的权重。这可能是因为你手动赋权——我们人类知道 *NYC* 和 *Big Apple* 常被视为同义词。希望我们的语义分析算法能依据 *apple* 与 *NYC* 在同一文档中共现的频率，自动识别它们的同义关系。

阅读代码清单 4.1 中为这三个主题及六个单词设定的加权和时，试着猜猜我们是如何得到这些权重的。你脑海中的语料库也许与我们不同，因此对这些单词的适当权重可能有不同看法。你可能如何改变它们？又能使用什么客观度量来评估这些比例（权重）？下一节我们将回答这个问题。

> **NOTE** 我们采用**带符号加权（signed weighting）**（允许对单词使用正权重与负权重）来生成主题向量。这使你可以为与主题“相反”的单词赋予负权重。由于手动操作，我们用易于计算的 **L1 范数** 对主题向量进行了归一化（即向量各维绝对值之和归一），而稍后在本章你将使用的真实 LSA 会用更有用的 **L2 范数** 来归一化，稍后我们会介绍不同范数及其距离。

在阅读这些向量时，你或许注意到单词与主题之间的关系可以“反转”。由三个主题向量组成的 3 × 6 矩阵可以转置，用以生成词汇表中每个单词的主题权重向量。下面这些权重向量即为六个单词的词–主题向量（word–topic vectors）：

```
>>> word_vector = {}
>>> word_vector['cat']  = .3 * topic['petness'] + \
...                       .1 * topic['animalness'] + \
...                       0 * topic['cityness']
>>> word_vector['dog']  = .3 * topic['petness'] + \
...                       .1 * topic['animalness'] + \
...                       .1 * topic['cityness']
>>> word_vector['apple']= 0 * topic['petness']  + \
...                      -.1 * topic['animalness'] + \
...                       .2 * topic['cityness']
>>> word_vector['lion'] = 0 * topic['petness']  + \
...                       .5 * topic['animalness'] + \
...                      -.1 * topic['cityness']
>>> word_vector['NYC']  =-.2 * topic['petness'] + \
...                       .1 * topic['animalness'] + \
...                       .5 * topic['cityness']
>>> word_vector['love'] = .2 * topic['petness']  + \
...                      -.1 * topic['animalness'] + \
...                       .1 * topic['cityness']
```

这六个词–主题向量（如图 4.1 所示），每个单词对应一个 3D 向量，表示该单词的语义。

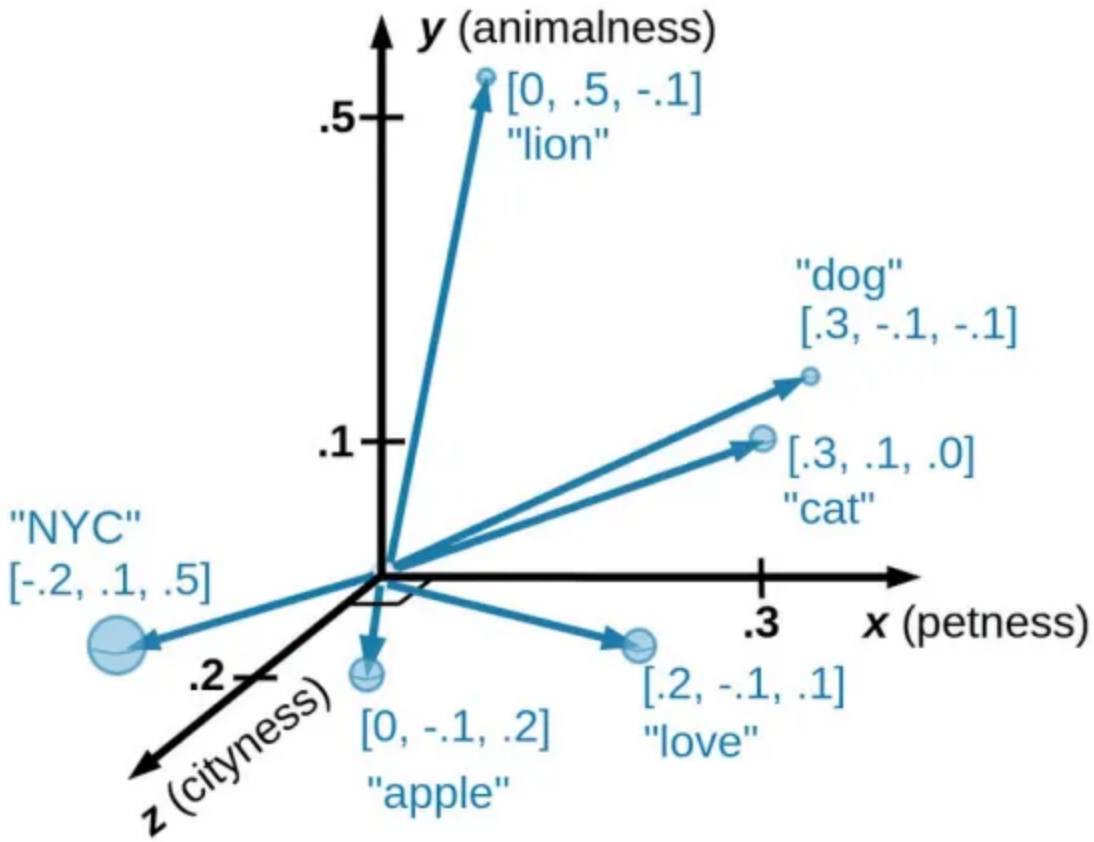

图 4.1　关于宠物与纽约市（NYC）的六个单词的 3D 向量思想实验

之前，每个主题向量都用单词及其权重线性组合而成，生成了六维向量，代表你的三个主题中单词的线性组合。现在，你手工构造了按主题表示文档的方法。如果你单纯统计这些六个单词的出现次数并乘以权重，就能得到该文档的三维主题向量。使用三维向量便于可视化；你可以绘制它们并以图形方式分享关于语料或特定文档的见解。另外，3D（或任何低维）向量空间非常适合机器学习分类问题，因为算法可以在向量空间用平面（或超平面）划分出类别。

虽然语料中的文档可能使用更多单词，此特定主题向量模型仅受这六个单词的影响。只要模型只需按三个维度或主题分隔文档，你的词汇量就可以无限增长。在这个思想实验中，你把六个维度（TF–IDF 归一化频次）压缩成三个维度（主题）。

这种主观、劳动密集型语义分析方法依赖人类直觉和常识来把文档拆分成主题，而常识难以编码进算法。显然，这并不适合机器学习流水线，而且无法很好地扩展到更多主题和单词。所以，让我们自动化这套手工过程——使用不依赖常识来为单词选择主题权重的算法。

每个加权和本质上都只是一次**点积（dot product）**；三个点积（加权和）不过是一次**矩阵乘法（matrix multiplication，即内积）**。你将一个 **3 × n** 权重矩阵与一个 TF–IDF 向量（文档中每个单词对应一个值）相乘，输出即为该文档的 **3 × 1** 主题向量。也就是说，你已完成从一个向量空间（TF–IDF）到另一个低维向量空间（主题向量）的**转换**。算法应创建一个 **n × m** 的矩阵（*n* 为词汇表大小，*m* 为主题数），可将文档的词频向量与之相乘，得到新的文档主题向量。

现在，假设你拥有若干不同主题的文档。如果把这些文档的表示转换到主题空间，便更容易按查询找出匹配文档：你能在高 **cityness** 的文档中寻找城市规划信息，在高 **petness** 的文档中寻找宠物护理建议。

但主题建模还可解决更基础的问题：识别语料中出现的趋势与主题。例如，对于一批开放式调查或在线产品评论，主题建模能帮助你识别受访者最常提到的问题。通过把评论中的词语连接到主题，如 *price*、*expensive*、*cost* 与 *taste*、*flavor*、*great*、*odd*，你可以找到聚焦于特定特性的评论。

4.1.4 主题评分算法

要确定这些主题向量，或从你已有的向量（如 TF–IDF 或 BOW 向量）推导出主题向量，仍然需要一种算法化的方法。机器本身无法判断哪些单词彼此相关，或它们各自代表什么，对吧？20 世纪英国语言学家 J. R. Firth 研究了如何估计一个单词或词素[5] 的意义。1957 年，他提出了计算单词主题的一条线索：

> **你可以通过一个词旁边的词来了解它的含义。**
> ——J. R. Firth

那么，怎样判定一个词的“同伴”呢？最直接的做法是在同一文档中统计共现，而第 3 章得到的 BOW 和 TF–IDF 向量正好可以做到这一点。这种“统计共现”的方法促成了若干算法的发展，用

来创建向量表示文档或句子内部的用词统计。

接下来几节，你将看到两种构造主题向量的算法。第一种是 **潜在语义分析（latent semantic analysis，LSA）**，它应用于你的 TF—IDF 矩阵，将单词聚合成主题。LSA 同样适用于 BOW 向量，但在 TF—IDF 向量上效果略好。LSA 通过优化主题，使主题维度保持多样性；使用这些新主题而不是原始单词时，仍能捕捉文档的大部分含义（语义）。为了表达文档含义，模型所需的主题数量远小于 TF—IDF 向量词汇表中的单词数量，因此 LSA 常被视为一种**降维技术**。LSA 减少了捕捉文档含义所需的维度数。[6]

另一种算法称为 **潜在狄利克雷分配（latent Dirichlet allocation，LDiA）**。由于本书用 LDA 来指代 *linear discriminant analysis*，因此将 *latent Dirichlet allocation* 缩写为 **LDiA**。

LDiA 在数学上沿着与 LSA 不同的方向前进，它使用非线性的统计算法来把单词归为一组。因此，与线性方法（如 LSA）相比，它通常需要更长的训练时间。这常使 LDiA 在许多实际应用中不太实用，且通常不应成为你的首选方法。不过，LDiA 生成的主题统计有时更贴近人类对单词和主题的直觉，因此 LDiA 主题往往更容易向老板解释。对于某些单文档任务（例如文档摘要），LDiA 也更有用。

在大多数分类或回归问题中，因其可扩展性和可解释性，你通常会选择 LSA。因此，本节将从 LSA 开始，并探究其所依赖的 SVD 线性代数。

4.2 挑战：检测有毒内容（Detecting toxicity）

要真正领略主题建模的威力，你将解决一个实际问题：识别 Wikipedia 评论中的毒性（toxicity）。**有毒评论（toxic comment）**指意在激怒、羞辱或恐吓其受众的信息。威胁、辱骂、粗俗语言和仇恨言论都是毒性的表现形式。这类毒性在维基这类参考网站上尤为危险，因为编辑者之间对话中的毒性可能蔓延到页面内容中。此外，Wikipedia 在一定程度上决定了整个互联网的基调，因为所有主流搜索引擎都信任它来回答大量问题，其中包括敏感和有争议的话题。

如果版主和平台管理员无法迅速侦测并阻断有毒内容的传播，社交媒体平台很快就会陷入毒性漩涡，毒性评论成为常态。负责任的社交媒体社区版主经常使用 LSA 来减少工作量，并降低易受伤害用户暴露于有毒内容的时间。与语言模型和其他深度学习 NLP 工具不同，LSA 计算效率很高，足以在最大的社交媒体平台上运行到网页规模。在本章中，你将使用一组 Wikipedia 讨论评论[7]，并自动将其分类为两类：**有毒**与**非有毒**。首先，使用下面的代码加载有毒评论数据集。

代码清单 4.2　有毒评论数据集　Python

```
>>> import pandas as pd
>>> pd.options.display.width = 120   # 让 pandas DataFrame 打印时多显示一些评论文本
>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/-/raw/master/'
...             'src/nlpia/data')
>>> url = DATA_DIR + '/toxic_comment_small.csv'
>>> comments = pd.read_csv(url)
>>> index = ['comment{}{}'.format(i, '!'*j) for (i, j) in
...          zip(range(len(comments)), comments.toxic)]
>>> comments = pd.DataFrame(
...     comments.values, columns=comments.columns, index=index)
>>> mask = comments.toxic.astype(bool).values
>>> comments['toxic'] = comments.toxic.astype(int)
>>> len(comments)
5000
>>> comments.toxic.sum()
650
>>> comments.head(6)
```

```
                      text  toxic
comment0  you have yet to identify where my edits violat...      0
comment1  as i have already said,wp:rfc on wp:ani. (...      0
comment2  your vote on wikiquote simple english when it ...      0
comment3  your stalking of my edits i've opened a thread...      0
comment4  straight from the smear site itself. the perso...      1
comment5  no, i can't see it either - and i've gone back...      0
```

你共有 5 000 条评论，其中 650 条被标记为二元类别标签 **toxic**。

在深入各种花哨的降维技术之前，先尝试用你已熟悉的向量表示 —— **TF-IDF** 来解决分类问题。但是，你将选择什么模型来分类这些信息呢？为此，让我们先看看下面代码中生成的 TF-IDF 向量。

代码清单 4.3　为有毒评论数据集创建 TF–IDF 向量　Python

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")

>>> def spacy_tokenize(sentence):
...     return [token.text for token in nlp(sentence.lower())]

>>> tfidf_model = TfidfVectorizer(tokenizer=spacy_tokenize)
>>> tfidf_docs = tfidf_model.fit_transform(
...     raw_documents=comments.text).toarray()
>>> tfidf_docs.shape
(5000, 19169)
```

spaCy 的分词器为你生成了 19 169 个词汇项。你拥有的词数几乎是消息数的 4 倍，是有毒评论数的近 30 倍。因此，模型可用的信息量有限，难以判断一条评论是否有毒。

你已经在本书早些章节（第 2 章）见过一种分类器：朴素贝叶斯。通常，当词汇量远大于数据集中标注实例数时，朴素贝叶斯效果不佳。因此，这一次我们需要采用不同的方案。

4.2.1　线性判别分析（LDA）分类器

在本章中，我们将介绍一种基于 LDA 的分类器。LDA 是你能找到的最快且最直接的模型之一，它所需的样本量也比更花哨的算法少。

LDA 的输入是带标签的数据，因此我们既需要表示消息的向量，也需要它们的类别标签。在本例中，我们有两类：**toxic** 评论和 **nontoxic** 评论。LDA 算法用到的部分数学超出了本书范围，但在二分类场景下，其实现相当直观。

本质上，当面对二分类问题时，LDA 算法会做以下事情：

1. 在你的向量空间中找到一条直线或一条轴，使得如果把空间中的所有向量（数据点）投影到这条轴上，两个类别尽可能分开。
2. 将所有向量投影到该直线上。
3. 根据两个类别之间的**切分点（cutoff point）**，预测每个向量属于其中一个类别的概率。

令人惊讶的是，在大多数情况下，能够最大化类别分离的那条线与连接各类别簇**质心**[8]的直线非常接近。下面我们手动近似执行一次 LDA，看看在数据集上的效果：

```
>>> mask = comments.toxic.astype(bool).values          # 选择有毒评论行
>>> toxic_centroid = tfidf_docs[mask].mean(axis=0)    # 计算有毒簇质心
>>> nontoxic_centroid = tfidf_docs[~mask].mean(axis=0)

>>> centroid_axis = toxic_centroid - nontoxic_centroid
>>> toxicity_score = tfidf_docs.dot(centroid_axis)
>>> toxicity_score.round(3)
array([-0.008, -0.022, -0.014, ..., -0.025, -0.001, -0.022])
```

toxicity_score 表示将每条向量在“非毒性质心 → 毒性质心”连线上的投影长度。你通过对每个 TF–IDF 向量与该连线向量做点积一次性计算了 5 000 个投影，这一向量化 NumPy 操作比 Python `for` 循环快上百倍。

接下来需把分数转换为实际的类别预测。理想情况下，分数应归一化到 0–1 区间，类似概率。有了归一化得分，你就能基于阈值推断分类——这里简单取 0.5。可使用 **sklearn** 的 `MinMaxScaler` 来完成归一化。

代码清单 4.4　基于毒性分数分类评论　Python

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> comments['manual_score'] = MinMaxScaler().fit_transform(
...     toxicity_score.reshape(-1, 1))
>>> comments['manual_predict'] = (comments.manual_score > .5).astype(int)
>>> comments[['toxic manual_predict manual_score'.split()]].round(2).head(
6)
           toxic  manual_predict  manual_score
comment0       0               0          0.41
comment1       0               0          0.27
comment2       0               0          0.35
comment3       0               0          0.47
comment4!      1               0          0.48
comment5       0               0          0.31
```

看起来不错！前六条消息几乎全部分类正确。接着检查在整个训练集上的表现：

```
>>> (1 - (comments.toxic - comments.manual_predict).abs().sum()
...     / len(comments))
0.895...
```

不错！用这套简单的“近似 LDA”方法，89.5 %的消息被正确分类。那么完整的 LDA 会怎样？我们使用 **scikit-learn** 中的 LDA 实现。

代码清单 4.5　LDA 在训练集上的表现　Python

```
>>> from sklearn.discriminant_analysis import \
...     LinearDiscriminantAnalysis
>>> lda_tfidf = LinearDiscriminantAnalysis(n_components=1)
>>> lda_tfidf = lda_tfidf.fit(tfidf_docs, comments['toxic'])
>>> comments['tfidf_predict'] = lda_tfidf.predict(tfidf_docs)
>>> float(lda_tfidf.score(tfidf_docs, comments['toxic']))
0.999...
```

这次得到了 99.9 %！几乎完美的准确率。这是否意味着你不需要更高级的主题建模算法，如 LDA 或深度学习?

这是个陷阱问题。你可能已经看出原因：我们没有划分测试集。这个 99.9 % 的接近满分成绩是在分类器已“见过”的“题目”上取得的——就像考试时遇到前一天复习过的原题。在真实世界的喷子和垃圾评论面前，该模型表现未必理想。

TIP　注意你在训练和预测中使用的类方法。**sklearn** 中的每个模型都提供诸如 `fit()` 和 `predict()` 之类的方法。许多分类器模型甚至还有 `predict_proba()` 方法，返回所有类别的概率得分。这样可在尝试解决机器学习问题时轻松替换不同模型算法，把脑力省下来做 NLP 工程师真正需要的创造性工作 —— 调整模型超参数，使其在现实世界中有效。

下面我们看看清单 4.4 中的代码在处理更接近现实的场景时的表现，你将把评论数据拆分成两部分：训练集与测试集（**sklearn** 里有相应函数）。然后观察分类器在未见过的消息上的表现。

代码清单 4.6　使用训练/测试拆分评估 LDA 模型性能　Python

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs,\
...     comments.toxic.values, test_size=0.5, random_state=271828)
>>> lda_tfidf = LinearDiscriminantAnalysis(n_components=1)
>>> lda = lda_tfidf.fit(X_train, y_train)        # 将线性判别分析（LDA）模型拟合到成千上万的特征上会耗费相当长的时间。请耐心等待——它正在用一个 2 万维的超平面切分你的向量空间!
>>> round(float(lda.score(X_train, y_train)), 3)
0.999
>>> round(float(lda.score(X_test, y_test)), 3)
0.554
```

使用词频—逆文档频率（TF—IDF）特征的模型在训练集上的准确率几乎完美，但在测试集上的准确率仅为 0.55——只比掷硬币略好。而真正有意义的是测试集准确率。这正是主题建模能够帮助

你的地方：它能让模型从较小的训练集泛化，以便在词汇组合不同（但主题相似）的消息上仍能良好工作。

TIP 请注意 `train_test_split` 中的 `random_state` 参数。`train_test_split()` 的算法具有随机性，每次运行都会得到不同的结果和准确率。如果希望流水线可复现，请为这些模型和数据集拆分器查找 seed 参数，并在每次运行时将种子设置为相同的值，从而获得可重复的结果。

让我们更深入地看看 LDA 模型的表现，使用一种称为混淆矩阵（confusion matrix）的工具。混淆矩阵会告诉你模型犯错的次数。错误分为两类：假阳性（false positive）和假阴性（false negative）。测试集中被标记为有毒但模型预测为无毒的样本属于假阴性，因为它们被错误地标记为阴性（无毒）而本应标记为阳性（有毒）。相反，测试集中无毒标签被错误标记为有毒的样本属于假阳性，因为它们本应标记为阴性（无毒）却被错误地标记为有毒。下面用 sklearn 函数来实现：

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_test, lda.predict(X_test))
array([[1261,  913],
       [ 201,  125]], dtype=int64)
```

嗯……仅凭这些数字并不容易看出问题所在。幸运的是，sklearn 考虑到你可能需要更直观的方式来向他人展示混淆矩阵，于是提供了专用函数。试试看：

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.metrics import ConfusionMatrixDisplay
>>> ConfusionMatrixDisplay.from_estimator(lda, X_test, y_test, cmap="Greys"
,
...     display_labels=['non-toxic', 'toxic'], colorbar=False)
>>> plt.show()
```

你可以在图 4.2 中看到生成的 matplotlib 图，展示了两种标签（有毒与无毒）的正确与错误预测次数。请查看该图，看看能否发现模型性能存在的问题。

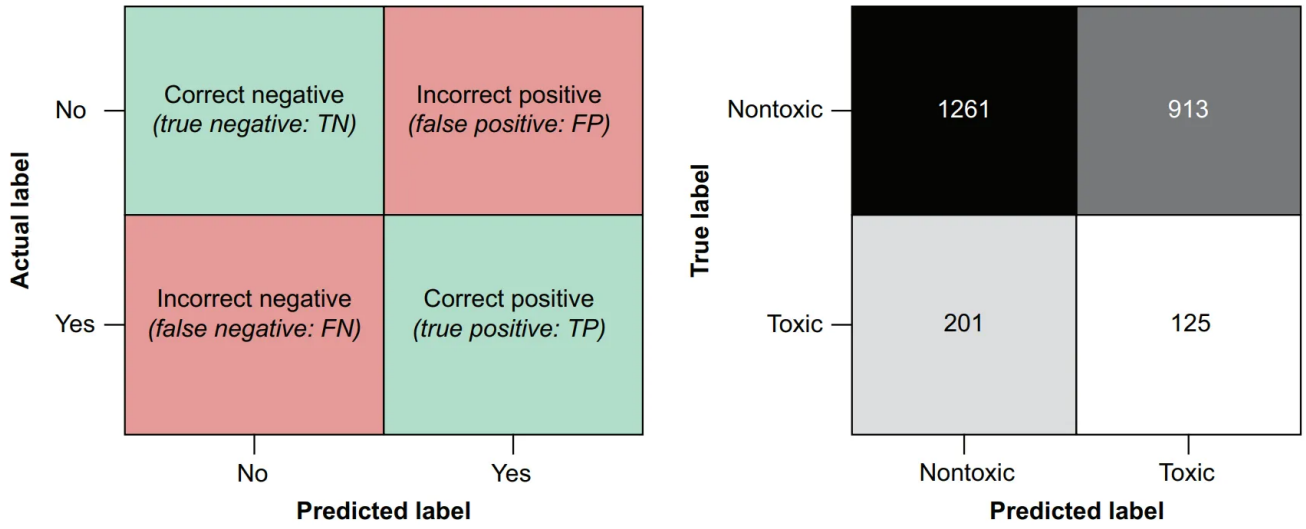

图 4.2　基于 TF–IDF 的分类器的混淆矩阵

首先，在测试集中实际有 326 条有毒评论中，模型只正确识别了 125 条——即 38.3%。这项度量（模型识别出我们关心类别实例的比例）称为召回率（recall，也称灵敏度 sensitivity）。另一方面，在模型标记为有毒的 1 038 条评论中，只有 125 条是真正的有毒评论，因此阳性标签仅在 12 %的情况下是正确的。这项度量称为精确率（precision）[9]。

现在你已经看到，精确率和召回率提供的信息比模型准确率更丰富。举例来说，如果你不用机器学习模型，而是使用确定性规则把所有评论都标记为无毒，由于数据集中约 13 %的评论实际上是有毒的，这个模型的准确率会达到 0.87——远高于你训练的上一版 LDA 模型！然而，它的召回率将为 0；在识别有毒消息的任务中丝毫不起作用。

你可能还会意识到这两项度量存在权衡。如果采用另一条确定性规则，把所有评论都标记为有毒，那么召回率将达到完美，因为所有有毒评论都会被正确分类。但精确率会很糟糕，因为被标记为有毒的大多数评论实际上完全正常。

根据不同的使用场景，你可能会决定优先考虑精确率或召回率。想象一下，你正在构建一个系统算法，用来检测评论中的毒性并提醒社区管理员，甚至在评论过于有毒时禁止用户发布。精确率过低意味着管理员被过度提醒，增加了工作负担。

如果你选择自动隐藏有毒评论，你那种过度提醒、低精确率的算法势必会惹恼用户，导致他们离开或想出创意策略绕过算法。另一方面，召回率过低则会增加真正有毒评论出现在平台上的概率。但在许多情况下，你希望精确率和召回率都保持在相当好的水平。

在这种情况下，你可能会使用 F1 分数（F1 score）——精确率和召回率的调和平均值。更高的精确率和更高的召回率都会提高 F1 分数，使得仅用一个指标就能更容易地对模型进行基准比较[10]。

你可以在附录 D 了解更多有关分析分类器性能的内容。现在我们仅记录一下该模型的 F1 分数，然后继续。

4.2.2 超越线性

LDA 在许多场景下都表现良好；然而，它仍有一些假设，当这些假设不成立时，分类器会表现欠佳。例如，LDA 假设所有类别的特征协方差矩阵相同。这是个相当强的假设！因此，LDA 只能学习类别之间的线性边界。

如果需要放宽这一假设，可以使用 LDA 的更一般形式——二次判别分析（QDA）。QDA 允许不同类别拥有不同的协方差矩阵，并分别估计每个协方差矩阵，因此能够学习二次（即弯曲的）边界[11]。这使得它更加灵活，在某些情况下性能更佳。

4.3 降维（reducing dimensions）

在深入讨论潜在语义分析（LSA）之前，让我们先从概念层面理解它对数据做了什么。LSA 在主题建模中的核心思想是降维（dimensionality reduction）。顾名思义，降维是一种寻找数据较低维表示的方法，同时尽可能保留信息量。

现在我们来仔细解析这一定义并理解其含义。为了帮助你建立直觉，让我们暂时离开自然语言处理（NLP），转而使用更直观的示例。首先，什么是数据的低维表示？想象把一个 3D 物体（比如你的沙发）表示在二维空间中。例如，在漆黑房间里，你在沙发后面打光，它投射在墙上的影子就是它的二维表示。

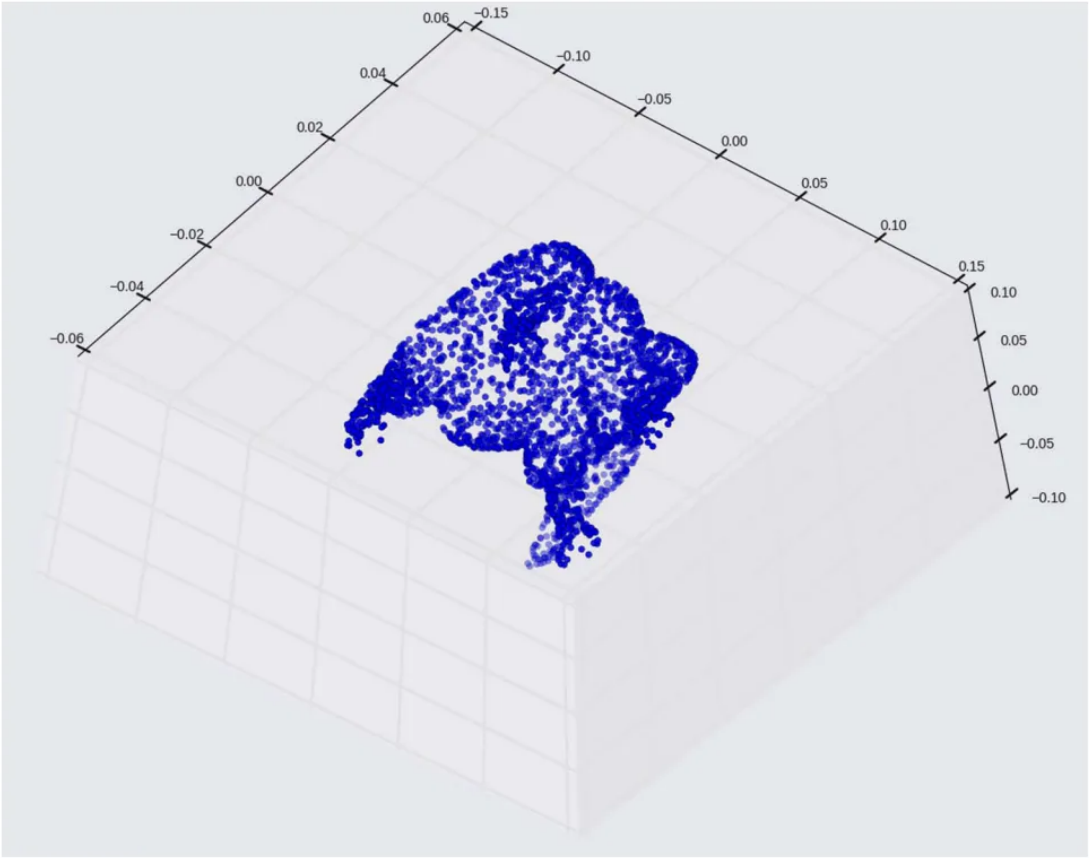

Figure 4.3 从下方“腹部”向上观察真实物体的点云（point cloud）

为什么我们需要这种表示？原因可能很多：也许我们没有能力按原样存储或传输完整数据，或者我们想把数据可视化以便更好地理解。你已经在讨论线性判别分析（LDA）时见识过可视化并聚类数据点的威力。但人脑实际上无法处理超过两三维的信息——而现实世界的数据，尤其是自然语言数据，往往具有数百甚至数千个维度。当我们想简化数据并对其进行可视化映射时，主成分分析（PCA，principal component analysis）等降维工具就非常有用。

另一个重要原因是我们在第 3 章简要提到的维度灾难（curse of dimensionality）。稀疏的多维数据更难处理，基于此训练的分类器也更容易过拟合。数据科学家经常使用的一条经验法则是：每个维度至少需要五条记录。即使对于小型文本数据集，TF−IDF 矩阵（TF−IDF）也能轻松达到一两万维，而许多其他类型的数据同样如此。

从沙发影子的例子可以看出，我们可以为同一数据集构建无限多种较低维度表示。其中一些表示优于其他表示，但在这里“更好”意味着什么？对于可视化数据，你可以直观地理解：能够让我们识别出物体的表示就优于不能的表示。例如，取一个真实物体的 3D 扫描点云，并将其投影到二维平面，结果见图 4.3。仅凭该表示，你能猜出原本的 3D 物体是什么吗？

继续我们的影子类比，想象正午的太阳照在一群人头顶上。每个人的影子都会是一个圆形斑块。我们能否仅根据这些斑块来判断谁高谁矮，或者谁留着长发？很可能不能。

现在你明白了，好的降维必须能够在新的表示中区分不同的对象和数据点；并且数据的并非所有特征（维度）对这种区分过程都同等重要。因此，有些特征可以轻松舍弃而不损失过多信息；但丢失某些特征则会显著削弱你理解数据的能力。由于这里涉及线性代数，你不仅可以选择舍弃或保留某个维度，还可以将多个维度组合成更小的维度集合，以更简洁的方式表示数据。下面让我们看看该如何做到这一点。

4.3.1 引入主成分分析（Principal Component Analysis, PCA）

现在你已经知道，要在更少的维度中表示数据，需要找到一组维度的组合，既能保留区分数据点的能力，又能让你，例如，将它们划分为有意义的聚类(cluster)。继续前面的影子示例，一个好的影子表示能够让你同时看到影子头部和腿部的位置。它通过保留这些对象之间的高度差来做到这一点，而不是像正午太阳下的影子那样把它们“压扁”到一个点。另一方面，我们身体的厚度从上到下大体一致，所以当你看到“扁平”的影子表示时，虽然丢掉了那一维，但损失的信息比丢掉高度时要少得多。在数学中，这种差异用方差(variance)来表示。仔细想想也合情合理：方差越大——即相对均值的偏离越大且越频繁——该特征就越有助于区分数据点。

不过，你不必只关注单个特征本身，特征之间的相互关系同样重要。在这里，可视化类比可能会开始失效，因为我们所处的三维空间是相互正交的，因此完全不相关。但回想前一部分中你见到的主题向量：动物属性(animalness)、宠物属性(petness) 和城市属性(cityness)。如果你检查这三者中每一对特征，很明显有些特征之间的关联比其他特征更强。大多数具有宠物属性的词语往往也具有动物属性。这种成对特征或维度的性质称为协方差(covariance)。它与相关系数(correlation) 密切相关，后者只是协方差除以这对特征各自的方差。协方差越高，特征之间的关联越强；因此，二者存在的冗余就越多，你可以由其中一个推断另一个。这也意味着，你可以找到一个单一维度来保留原本这两个维度中绝大部分的方差信息。

总而言之，为了在不丢失信息的前提下减少描述数据的维度，你需要找到一种表示方式，使得新的各坐标轴上的方差最大化，同时降低各维度之间的依赖关系，并去除那些协方差较高的维度。这正是主成分分析（PCA）所做的事情：它寻找一组方差最大的维度。这些维度是正交归一(orthonormal) 的（类似物理世界中的 x 轴、y 轴、z 轴），称为主成分(principal components)——方法名亦由此而来。PCA 还允许你查看每个维度所贡献的方差大小，从而选择能保留数据集“本质”的最优主成分数量；随后它会把你的数据投影到这一组新的坐标系中。

在深入了解 PCA 如何实现这一点之前，先来直观感受一下它的魔力。下面的代码清单将使用 scikit-learn 的 PCA 方法，对你在图 4.3 中看到的同一个 3D 点云进行处理，找出能够最大化该点云方差的两个维度。

代码清单 4.7 PCA 魔法 Python

```
>>> import pandas as pd
>>> from sklearn.decomposition import PCA
>>> import seaborn
>>> from matplotlib import pyplot as plt

>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/'
...             '-/raw/master/src/nlpia/data')

>>> df = pd.read_csv(DATA_DIR + '/pointcloud.csv.gz', index_col=0)
>>> pca = PCA(n_components=2)
>>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy'))
>>> df2d.plot(kind='scatter', x='x', y='y')
>>> plt.show()
```

当你把 3D 点（向量）的维度降到 2D 时，就像给那团 3D 点云拍了一张照片。结果可能看起来像图 4.4 左或右的图片，但它永远不会倾斜或旋转到新的角度。x 轴（轴 0）始终与点云最长的轴对齐，也就是点分布最广的方向。这是因为 PCA 总是寻找能够最大化方差的维度，并按方差从大到小的顺序排列它们。方差最大的方向将成为第一轴（x）；方差第二大的方向在 PCA 变换后成为第二轴（y）。不过这些坐标轴的正负方向（极性）是任意的，优化过程可以将向量（点）绕 x 轴、y 轴或二者同时镜像（翻转）。

现在我们已经见识过 PCA 的实际效果了[12]，接下来看看它是如何找到这些主成分的，使我们能够在更少的维度中处理数据而又不丢失太多信息。

4.3.2 奇异值分解（Singular Value Decomposition, SVD）

在 PCA 的核心是一种称为奇异值分解（Singular Value Decomposition, SVD）13 的数学过程。SVD 是一种算法，可将任意矩阵分解为三个因子——三 个矩阵，使它们相乘即可重建原矩阵。这类似于为一个大整数找到恰好三个整数因子，但此处的因子不是标量整数，而是具有特殊性质的二维实数矩阵。

假设我们的数据集由 m 个 n 维点组成，用矩阵 **W** 表示。在完整形式下，**W** 的 SVD 用数学符号表示（假设 m > n）为：

$$W_{m\times n} = U_{m\times m} \cdot S_{m\times n} \cdot V^T_{n\times n} \tag{4.1}$$

矩阵 **U**、**S**、**V** 具有特殊性质：**U** 和 **V** 为正交（orthogonal）矩阵，与各自的转置相乘会得到单位矩阵；**S** 为对角（diagonal）矩阵，只有对角线上元素可能为非零。

请注意等号：如果将 U、S、V 相乘即可得到完全相同的 W，即原始的 TF−IDF 向量矩阵，每行对应一个文档。然而这些矩阵的最小维度仍为 n，而你想降低维度数量。因此本章我们使用 SVD 的简化或截断形式（truncated SVD）14，只保留感兴趣的前 p 个维度。

你或许会问：“能否先做完整 SVD，再选方差最大的维度保留？”理论上完全可行——这正是 SVD 的本质。但使用截断 SVD 还有其他优势：存在多种算法可以高效地对大型矩阵进行截断 SVD 分解，尤其在矩阵是稀疏（sparse）时更明显。稀疏矩阵指多数单元值相同（通常为 0 或 NaN）。NLP 的词袋（BOW）和 TF−IDF 矩阵通常是稀疏的，因为大多数文档不包含词表中的许多词。

截断 SVD 的数学形式为：

$$W_{m\times n} = U_{m\times p} \cdot S_{p\times p} \cdot V^T_{n\times p} \tag{4.2}$$

公式中 m、n 分别为原矩阵的行数和列数；p 为保留的维度数。以马形点云示例为例，若要在二维空间显示点云，则 p=2。此时矩阵 V 含有二维行向量，每个向量表示一个文档的 m 维 TF−IDF 向量，用于绘制二维散点图。

Scikit-learn 的 `TruncatedSVD.fit()` 可近似求解上述公式，当 p=2 时生成 V 中的二维向量。若将 SVD 用于 LSA（潜在语义分析），p 即希望提取的主题数。p 必须小于 m 和 n：你不可能创建比词汇表大小 n 更多的主题，也不能多于语料库文档数 m。

由于使用近似等号，我们无法期望将因子相乘后得到完全相同的矩阵；信息必有损失。收获则是：在更少维度中表示数据。马形点云无需绘制庞大的三维图，也能传达“马形”本质；实际应用中，PCA 可将上百、上千维的数据简化为更短向量，便于分析、聚类和可视化。

U、S、V 分别有什么用？先给出直观解释，更深入的应用将在下一章介绍 LSA 时探讨。

先看 VT（以及其转置 V）。矩阵 V 的列有时称为主方向（principal directions）、也称主成分（principal components）。本章沿用 scikit-learn 的主成分命名。

可将 V 视为“转换器”，把数据从“旧”空间（矩阵 W 所在的世界）映射到新的低维空间。

若在三维马形点云中再加入一些点，想知道它们在二维表示中的位置，而不必重算所有点的变换，只需将每个新点 q 乘以 V 即可：

$$\hat{q} = q \cdot V$$

那么 $U \cdot S$ 表示什么？经过一些代数推导，它便是已映射到新空间的数据！即你的数据点在新的、更低维度表示中的坐标。

4.4 潜在语义分析（Latent Semantic Analysis, LSA）

终于，我们可以不再“玩马”游戏了，回到主题建模！让我们看看之前学到的降维、PCA 和 SVD 在文本数据中寻找主题和概念时是如何发挥作用的。

先从数据集本身开始。你将使用 4.1 小节中用于 LDA 分类器的同一份评论语料库，并通过 TF–IDF 转换成矩阵。你可能记得，这个结果被称为 **词–文档矩阵**（term–document matrix）。这个名字很有用，因为它让你直观地理解矩阵的行和列分别包含什么：行包含词项——你的词汇表单词，列包含文档。

让我们重新运行代码清单 4.1 和 4.2，再次得到 TF–IDF 矩阵。在深入 LSA 之前，我们查看一下矩阵的形状：

```
>>> tfidf_docs.shape
(5000, 19169)
```

这里是什么？这是一个 19 169 维的数据集，其空间由语料库词汇表中的词项定义。在这个空间里用单个向量表示每条评论相当麻烦，因为每个向量里有近 20 000 个数——比消息本身还多！并且很难判断消息或其中的句子在概念上是否相似——例如表达 *leave this page* 和 *go away* 的向量相似度会非常低，尽管它们的含义几乎相同。因此，用 TF–IDF 表示时，对文档进行聚类和分类要困难得多。

还要注意，在你的 5 000 条消息中只有 650 条被标注为有毒（toxic），占 13%。因此训练集极度不平衡，非毒性评论与毒性评论之比约为 8:1（人身攻击、淫秽、种族诽谤等）。此外，你有一个巨大的词汇表——词汇表标记数为 25 172，反而大于 4 837 条消息（样本）数量。所以，你在词汇表（或词典）中拥有的唯一单词远多于评论条数——而与毒性评论相比更是如此。这是过拟合[15] 的配方。

过拟合意味着你只会依赖词汇表中的少数几个词。因此，毒性过滤器将取决于那些毒性词出现在过滤掉的消息中。这导致喷子只需为这些毒性词寻找同义词即可绕过过滤器。如果你的词汇表不包含这些新同义词，过滤器就会把那些巧妙构造的评论误判为非毒性。

过拟合是 NLP 中的固有问题。很难找到一个自然语言标注数据集，能够覆盖所有可能导致同一标签的变体。我们无法找到包含人们表达毒性和非毒性方式所有差异的理想评论集，只有少数公司有资源去创建这样的数据集。因此，我们需要对抗过拟合的对策：使用能够在很少样本上泛化良好的算法。

对抗过拟合的主要策略是将数据映射到一个新的低维空间。这个新空间由词语或**主题**（topic）的加权组合定义，你的语料库以多种方式谈论这些主题。用主题而不是具体的词频来表示消息，将使你的 NLP 流水线更通用，让垃圾信息过滤器能处理更广泛的消息。LSA 正是这样做的——它使用上一节发现的 SVD 方法，找到新的主题维度，使方差最大化。

这些新主题不一定与人类理解的主题（如 *pets* 或 *history*）一一对应。机器并不理解哪些词组合表示什么，只知道它们经常一起出现。当它看到 *dog*、*cat*、*love* 经常同时出现时，就把它们放到一个主题里，但并不知道这是关于宠物的话题；主题里还可能包含 *domesticated*、*feral* 等意思相反的词。如果它们在同一文档中经常一起出现，LSA 会给它们在同一主题上较高的权重。给主题命名是人类的工作。

不过，你不必给主题命名才能使用它们。正如你不需要了解词干化 BOW 向量或 TF–IDF 向量中成千上万维的含义一样，你也不需要知道所有主题的含义。你仍可以用这些新主题向量做向量运算，像处理 TF–IDF 向量那样相加、相减、估计文档间基于主题表示的相似度，而这种相似度更准确，因为新的表示真正考虑了词语的意义及其与其他词语的共现关系。

4.4.1 深入语义分析

说 LSA 够多了——让我们写点代码！这次，我们将使用另一个 scikit-learn 工具 `TruncatedSVD`，它执行上一节介绍的截断 SVD 方法。我们也可以使用 PCA，但 `TruncatedSVD` 针对稀疏矩阵进行了优化，对 TF–IDF 和 BOW 矩阵效果更佳。

我们将先把维度从 9 232 减少到 16。稍后再解释为什么选择这个数字。

代码清单 4.8 使用 TruncatedSVD 进行 LSA　　Python

```
>>> from sklearn.decomposition import TruncatedSVD
>>>
>>> svd = TruncatedSVD(n_components=16, n_iter=100)       # TruncatedSVD 内部的 SVD 算法采用随机化，因此我们迭代 100 次以抵消随机性
>>> columns = ['topic{}'.format(i) for i in range(svd.n_components)]
>>> svd_topic_vectors = svd.fit_transform(tfidf_docs)      # fit_transform 一步完成 TF–IDF 向量的分解并转换为主题向量
>>> svd_topic_vectors = pd.DataFrame(svd_topic_vectors, columns=columns,
...                                  index=index)
>>> svd_topic_vectors.round(3).head(6)
```

通过 `fit_transform` 方法，你已经得到了新表示中的文档向量。现在，每条评论不再用 19 169 维频率计数表示，而是仅用 16 维表示。这个矩阵也称为 **文档–主题矩阵**。查看列即可看到每条评论在各主题上的“表达”程度。

NOTE 这些方法与矩阵分解过程的关系：`fit_transform` 的结果正是 $U \cdot S$ ——你的 TF–IDF 向量投影到新空间，而矩阵 V 则保存在 `TruncatedSVD` 对象的 `components_` 属性中。

如果你想探索主题，可以查看每个词或词组在各主题中的权重。首先，需要把词分配到转换后的所有维度。注意调整顺序，因为 `TfidfVectorizer` 把词汇表存为字典，键是词，值是索引：

```
>>> list(tfidf_model.vocabulary_.items())[:5]          # 将词汇表变成可迭代对象，items() 方法列出前五项
[('you', 18890),
 ('have', 8093),
 ('yet', 18868),
 ('to', 17083),
 ('identity', 8721)]
>>> column_nums, terms = zip(*sorted(zip(tfidf_model.vocabulary_.values(),
...                                      tfidf_model.vocabulary_.keys())))
 # 按词频索引排序词汇表；zip(*sorted(zip())) 的模式在按非首元素排序后再 unzip 很有用
>>> terms[:5]
('\n', '\n ', '\n \n', '\n \n ', '\n ')
```

现在，你可以创建一个漂亮的 pandas `DataFrame`，把所有列和行放到正确位置。但显然，我们的前几个词只是不同组合的换行符——不太有用！

数据集提供者应该先清洗一下数据。使用 pandas 的 `DataFrame.sample()` 方法，随便查看一些词汇表中的随机单词：

```
>>> topic_term_matrix = pd.DataFrame(
...     svd.components_, columns=terms,
...     index=['topic{}'.format(i) for i in range(16)])
>>> pd.options.display.max_columns = 8
>>> topic_term_matrix.sample(5, axis='columns',
...                          random_state=271828).head(4)  # 使用固定 random_state 保证输出一致
```

这些单词看起来都不是天生的有毒词。让我们看看一些直觉上应该出现在有毒评论中的词，并查看它们在不同主题上的权重：

```
>>> pd.options.display.max_columns = 8
>>> toxic_terms = topic_term_matrix[
...     'pathetic crazy stupid idiot lazy hate die kill'.split()
... ].round(3) * 100           # 乘以 100 让权重更易读和比较
>>> toxic_terms
          pathetic  crazy  stupid  idiot  lazy  hate  die  kill
topic0        0.3    0.1     0.7    0.6   0.1   0.4  0.2   0.2
topic1       -0.2    0.0    -0.1   -0.3  -0.1  -0.4 -0.1   0.1
topic2        0.7    0.1     1.1    1.7   0.0   0.9  0.6   0.8
topic3       -0.3   -0.0    -0.0    0.0   0.1  -0.0  0.0   0.2
topic4        0.7    0.2     1.2    1.4   0.3   1.7  0.6   0.0
topic5       -0.4   -0.1    -0.3   -1.3  -0.1   0.5 -0.2  -0.2
topic6        0.0    0.1     0.8    1.7  -0.1   0.2  0.8  -0.1
>>> toxic_terms.T.sum()
topic0     2.4
topic1    -1.2
topic2     5.0
topic3    -0.2
topic4     5.9
topic5    -1.8
topic6     3.4
topic7    -0.7
topic8     1.0
topic9    -0.1
topic10   -6.6
...
```

主题 2 和主题 4 似乎更可能包含毒性倾向，而主题 10 看起来是“反毒性”主题。因此，与毒性相关的词在某些主题中权重大、在其他主题中权重小；并不存在单一明显的毒性主题编号。

`transform` 方法所做的只是将传入的数据乘以 V 矩阵，它保存在 `components_` 属性中。想验证的话，可以打开 `TruncatedSVD` 的源码亲眼看看！[16]

4.4.2 TruncatedSVD 还是 PCA?

此时，你可能会问自己：“为什么在马形示例中我们用的是 scikit-learn 的 `PCA` 类，而在评论数据集的主题分析中却用 `TruncatedSVD`？我们不是说过 PCA 本质上也是基于 SVD 算法吗？”的确如此——如果你查看 sklearn 中 `PCA` 和 `TruncatedSVD` 的源码，会发现两者的大部分实现几乎一模一样，都使用同样的 SVD 分解算法。不过，它们之间仍有一些差异，使得在某些场景下各自更具优势。

最大的区别在于，**`TruncatedSVD` 在分解之前不会对矩阵做居中处理，而 `PCA` 会**。因此，如果要用 `TruncatedSVD` 又想让数据居中，可以手动把每一列的均值减掉，例如：

```
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean()
```

对比一下在居中数据上跑 TruncatedSVD 与直接跑 PCA，两种方法的结果将完全一致，建议你亲自试试！

对于稀疏的 TF—IDF 矩阵，居中往往会让稀疏矩阵变得稠密，导致计算更慢、内存占用更高。PCA 更适合处理稠密矩阵，且能对小矩阵计算精确的完整 SVD；而 TruncatedSVD 假定输入本身就是稀疏矩阵，采用更快的近似随机化算法，因此处理 TF—IDF 数据的效率要高得多。[17]

4.4.3 LSA 在毒性检测上的表现如何？

研究主题够久了——现在来看看用 16 维表示的评论在分类任务中表现如何。我们重用清单 4.3 的代码，但输入改为新的 16 维向量：

```
>>> X_train_16d, X_test_16d, y_train_16d, y_test_16d = train_test_split(
...     svd_topic_vectors, comments.toxic.values, test_size=0.5,
...     random_state=271828)
>>> lda_lsa = LinearDiscriminantAnalysis(n_components=1)
>>> lda_lsa = lda_lsa.fit(X_train_16d, y_train_16d)
>>> round(float(lda_lsa.score(X_train_16d, y_train_16d)), 3)
0.881
>>> round(float(lda_lsa.score(X_test_16d, y_test_16d)), 3)
0.880
```

效果惊人！训练集准确率从 TF—IDF 的 99.9 % 降到 88.1 %，但测试集准确率却提升了 33 %——改进显著。

接着看 F1 分数：

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_test_16d, lda_lsa.predict(X_test_16d))
0.342
```

与 TF—IDF 向量分类相比，F1 几乎翻倍，表现不错。

实际项目中常需比较多种模型，手动翻阅结果既麻烦又易错，因此数据科学家会把模型参数和评测指标记录进**超参数表**（hyper-parameter table）。下面动手实现：

```
>>> hparam_table = pd.DataFrame()
>>> tfidf_performance = {'classifier': 'LDA',
...     'features': 'TF-IDF (spacy tokenizer)',
...     'train_accuracy': 0.99,
...     'test_accuracy': 0.554,
...     'test_precision': 0.383,
...     'test_recall': 0.12,
...     'test_f1': 0.183}
>>> hparam_table = hparam_table.append(tfidf_performance, ignore_index=True
)  # ignore_index 让 DataFrame 重新编号
```

把提取指标的流程封装成函数更便捷。

代码清单 4.9 在超参数表中写入一条记录的函数 Python

```
>>> from sklearn.metrics import precision_score, recall_score, f1_score

>>> def hparam_rec(model, X_train, y_train, X_test, y_test,
...                model_name, features):
...     return {
...         'classifier': model_name,
...         'features': features,
...         'train_accuracy': float(model.score(X_train, y_train)),
...         'test_accuracy': float(model.score(X_test, y_test)),
...         'test_precision': precision_score(y_test, model.predict(X_test
)),
...         'test_recall': recall_score(y_test, model.predict(X_test)),
...         'test_f1': f1_score(y_test, model.predict(X_test))
...     }

>>> lsa_performance = hparam_rec(lda_lsa, X_train_16d, y_train_16d,
...     X_test_16d, y_test_16d, 'LDA', 'LSA (16 d)')
>>> hparam_table = hparam_table.append(lsa_performance, ignore_index=True)
>>> hparam_table.T     # 转置表格方便查看
```

你还可以再进一步，把大部分流程包装成一个 `evaluate_model` 函数，日后无需复制粘贴。

4.4.4 其他降维方式

SVD 无疑是最流行的数据集降维方法，这也使得在考虑主题建模时，LSA 通常是首选。不过，还有若干其他降维技术也能达到同样目标，并非所有方法都会用于 NLP，但了解它们总是有益。本节提到两种方法：随机投影（random projection）和非负矩阵分解（nonnegative matrix factorization, NMF）。

随机投影是一种将高维数据投射到低维空间的方法，旨在保持数据点之间的距离。其随机（stochastic）特性使其更易于在并行计算环境中运行；同时，它占用的内存更少，因为不像 PCA 那样需要一次性把所有数据加载到内存。由于计算复杂度更低，随机投影在分解速度是关键因素、且维度极高的数据集上偶尔会使用。

NMF 是另一种与 SVD 相似的矩阵分解方法，但它假定数据点和分量均为非负（nonnegative）。NMF 更常见于图像处理和计算机视觉领域，但在 NLP 和主题建模中也偶尔派上用场。在大多数情况下，沿用底层使用久经考验的 SVD 算法的 LSA 更为稳妥。

4.5 潜在狄利克雷分配（Latent Dirichlet Allocation, LDiA）

本章的大部分内容都在探讨潜在语义分析及使用 scikit-learn 将词语和短语表示为向量的各种方法。LSA 应成为大多数主题建模、语义搜索或基于内容的推荐引擎的首选工具。[18] 其数学原理简单高效，能够在几乎不降低精度的情况下将线性变换应用于新的自然语言批次且无需重新训练。接下来，你将学习一种更复杂的算法——潜在狄利克雷分配（LDiA），在某些场景中它能取得略好的效果。

LDiA 与 LSA（底层同样基于 SVD）在构建主题模型时做了许多相似的事情，但与 LSA 不同，LDiA 假设词频服从狄利克雷（Dirichlet）分布。因此，在把词分配到主题时，它在统计上比 LSA 的线性数学更精确。

LDiA 使用与你在本章思考实验中类似的方式，创建语义向量空间模型（即主题向量）。在思考实验中，你按单词在同一文档中出现的频率手动将其分配到主题。对于给定文档，主题的混合可以通过该文档中各主题词的分布与共现来确定。由于分配到主题的单词与分配到文档的主题更易于理解，LDiA 的主题模型通常比 LSA 更符合直觉。

LDiA 假设每个文档是若干主题的混合（线性组合），主题数量由你在训练 LDiA 模型时指定；同时假设每个主题可由词频分布表示。文档中每个主题的概率（或权重）以及单词被分配到主题的概率都假定服从狄利克雷先验（prior）分布，这也是算法名称的由来。

4.5.1 LDiA 的思路

LDiA 方法最初为生物学应用而开发，后来才被引入 NLP。斯坦福的研究者[19] 通过“反转”思考实验提出了该思路：设想有一台机器只做随机数生成，便可写出比人类语料还大的文档集合，并分析这些文档统计特征。

他们假想的机器只需做两个决定即可开始为某个文档生成词袋：机器像掷骰子一样，以某种概率分布随机选择词语，类似给骰子指定面数并组合多颗骰子来得到想要的概率分布。为了让生成的单词数与主题数分布与真实文档匹配，需要为不同值设定特定的概率分布。

掷两次“骰子”代表：

1. 生成该文档的单词数量（泊松 Poisson 分布）
2. 将多少个主题混合到该文档中（狄利克雷 Dirichlet 分布）

得到这两个数字后，难点在于为文档选择词语。想象中的 BOW 生成机器在所有主题间迭代，随机选取与主题匹配的词，直到命中步骤 1 决定的词数。确定哪些词属于哪些主题（即词语对主题的适切性）是难点，但一旦确定，你的“机器人”只需从词–主题概率矩阵中查表即可。如果忘了该矩阵长什么样，可以回顾本章前面的简单示例。

这台机器只需一个参数（步骤 1 中泊松分布的参数）来指定平均文档长度，再加上几个参数定义狄利克雷分布以设定主题数。接着，文档生成算法需要一张词–主题概率矩阵（词汇表）以及一组它喜欢“谈论”的主题混合。

现在，把文档生成（写作）问题反转为最初的问题：从现有文档估计主题和词语。你需要度量或计算步骤 1、2 中关于词语和主题的参数，然后根据文档集合计算词–主题矩阵。这正是 LDiA 所做的。

Blei 与 Ng 认识到，通过分析语料中各文档的统计信息，可以确定步骤 1 和 2 的参数。例如，对步骤 1，他们可以计算语料中所有 BOW（或 n–gram）的平均单词数，如下所示：

```
>>> total_corpus_len = 0
>>> for document_text in comments.text:
...     total_corpus_len += len(spacy_tokenize(document_text))
>>> mean_document_len = total_corpus_len / len(comments.text)
>>> round(mean_document_len, 2)
54.21
```

或者使用 `sum` 函数，写成下面这样：

```
>>> sum((len(spacy_tokenize(t)) for t in comments.text)
... ) * 1. / len(comments.text)
54.206
```

请牢记，必须直接从词袋（BOW）来计算这一统计量。要确保你在计数字符串前，已经将文档中的词语进行了分词和向量化，并完成了停用词过滤或其他规范化操作；而且只统计 BOW 真正使用的唯一词项。因此，本 LDiA 算法依赖于 BOW 向量空间模型，而不是像 LSA 那样接受 TF–IDF 矩阵作为输入。

你需要为 LDiA 模型指定的第二个参数——**主题数**——要棘手一些。在你把词分配到主题之后，才能直接度量某个数据集所需的主题数。和 *k*-means、k 最近邻以及其他聚类算法一样，你必须事先告诉 LDiA 要找多少主题（类似 *k*-means 中的 *k* 值，即簇数）。可以先猜一个主题数，再检查它是否适用于你的文档集。一旦告诉 LDiA 要寻找多少个主题，它就会在每个主题中寻找合适的词汇混合，以优化其目标函数[20]。

可以通过调节这个超参数 *k*（主题数）[21] 来优化它；如果你能衡量 LDiA 语言模型对文档语义表示的质量，还可以自动完成这一过程。用来优化的一个“代价函数”示例是：在某个分类或回归任务（如情感分析、文档关键词标注或主题分析）中，模型的表现有多好。只需要一些带标签的文档来测试你的主题模型或分类器即可。

4.5.2 针对评论的 LDiA 主题模型

LDiA 生成的主题通常更易于人类理解与解释。这是因为经常一起出现的词会被分配到同一主题，而人们也会期望如此。LSA 倾向于保持最初彼此距离较远的向量分散，而 LDiA 则倾向于把最初很接近的词进一步聚拢。

这听起来好像是同一回事，但实际上并非如此——二者的数学目标函数不同。为了在较低维空间内保持高维向量彼此距离，LDiA 必须以非线性方式对空间（以及向量）进行扭曲和拉伸。要直观展示这种效果颇为困难，除非在 3D 中做完投影，再把结果向量投影到 2D。

下面看看它在有毒评论数据集上的表现。首先计算 TF–IDF 向量，然后为每条评论（文档）计算一些主题向量。和之前一样，我们假设只使用 16 个主题（组件）来分类评论的毒性；保持主题数较少有助于减轻过拟合[22]。LDiA 使用原始 BOW 计数向量，而不是归一化后的 TF–IDF 向量。第三章已演示过此过程，下面清单再次展示。

代码清单 4.10 计算评论数据集的 BOW 向量 Python

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> counter = CountVectorizer(tokenizer=spacy_tokenize)
>>> bow_docs = pd.DataFrame(
...     counter.fit_transform(
...         raw_documents=comments.text).toarray(), index=index)
>>> column_nums, terms = zip(*sorted(zip(counter.vocabulary_.values(),
...                                      counter.vocabulary_.keys())))
>>> bow_docs.columns = terms
```

让我们核对一下第一个评论 `comment0` 的计数是否合理：

```
>>> comments.loc['comment0'].text
'you have yet to identify where my edits violated policy.
 4 july 2005 02:58 (utc)'
>>> bow_docs.loc['comment0'][bow_docs.loc['comment0'] > 0].head()
you       1
(         1
)         1
.         1
02:58     1
Name: comment0, dtype: int64
```

接下来，我们像将 LSA 应用于 TF–IDF 矩阵一样，把 LDiA 应用于计数向量矩阵：

```
>>> from sklearn.decomposition import LatentDirichletAllocation as LDiA
>>> ldia = LDiA(n_components=16, learning_method='batch')   # LDiA 处理大量主题与大词表时比 PCA 或 SVD 慢一些
>>> ldia = ldia.fit(bow_docs)
>>> ldia.components_.shape
(16, 19169)
```

你的模型已把 19 169 个词项分配到 16 个主题（组件）。下面查看前几个词以及它们的分配情况（与你的输出可能不同，因为 LDiA 是随机算法，需要固定随机种子才能复现）。

```
>>> pd.set_option('display.width', 75)
>>> term_topic_matrix = pd.DataFrame(ldia.components_, index=terms,
...                                  columns=columns)
>>> term_topic_matrix.round(2).head(3)
```

（这是之前为 LSA 主题模型构建的同一矩阵的转置！）

看起来 LDiA 主题向量中的值比 LSA 的跨度要大得多——既有接近零的数，也有非常大的数。我们用与 LSA 相同的技巧，看看典型有毒词在每个主题中的权重。

代码清单 4.11 不同 LDiA 主题中有毒词的分布 Python

```
>>> toxic_terms = term_topic_matrix.loc['pathetic crazy stupid lazy idiot hate
...                                     die kill'.split()].round(2)
>>> toxic_terms
```

结果与 LSA 表示差异明显：某些词在部分主题中权重大、在另一些主题中则权重极小。`topic0` 和 `topic1` 对有毒词几乎“无感”，而 `topic2` 与 `topic15` 对其中至少四五个有毒词的权重相当大，`topic14` 对词 **hate** 的权重尤其高。

让我们看看在该主题中得分较高的其他词。由于此前未对数据集做任何预处理，许多词并无意义。聚焦字母字符且超过三个字母的词，可以删掉大量停用词：

```
>>> non_trivial_terms = [term for term in term_topic_matrix.index
...                      if term.isalpha() and len(term) > 3]
>>> term_topic_matrix.topic14.loc[non_trivial_terms].sort_values(ascending
=False)[:10]
hate        480.062500
killed       14.032799
explosion     7.062500
witch         7.033359
june          6.676174
wicked        5.062500
dead          3.920518
years         3.596520
wake          3.062500
arrived       3.062500
```

可以看到，这些词之间在语义上彼此相关：*killed*、*hate*、*wicked*、*witch* 等似乎都属于 **毒性** 领域，即使只做快速查看，也能发现词与主题的分配是可以理解和推理的。

在拟合分类器之前，需要为所有文档（评论）计算这些 LDiA 主题向量。比较一下它们与同样文档的 LSA 主题向量的不同之处：

```
>>> ldia16_topic_vectors = ldia.transform(bow_docs)
>>> ldia16_topic_vectors = pd.DataFrame(ldia16_topic_vectors,
...                                     index=index, columns=columns)
>>> ldia16_topic_vectors.round(2).head()
```

可以看到，这些主题之间分离得更清晰：在分配给消息的主题中，有大量的零值。正因为如此，当你需要基于 NLP 流水线结果做业务决策时，LDiA 主题更易于向同事解释。

那么，LDiA 主题对人类来说不错，但对机器呢？基于这些主题，你的 LDA 分类器表现如何？

4.5.3 使用 LDiA 检测毒性

让我们看看这些 LDiA 主题在预测实用任务（例如评论毒性）方面表现如何。你将再次使用 LDiA 主题向量来训练一个 LDA 模型——就像之前分别用 TF–IDF 向量和 LSA 主题向量训练过两次一样。借助代码清单 4.5 中编写的快捷评估函数，只需几行代码即可评估模型：

```
>>> model_ldia16 = LinearDiscriminantAnalysis()
>>> ldia16_performance = evaluate_model(ldia16_topic_vectors,
...     comments.toxic, model_ldia16, 'LDA', 'LDIA (16 components)')
>>> hparam_table = hparam_table.append(ldia16_performance,
...     ignore_index=True)
>>> hparam_table.T
```

从结果来看，基于 16 个 LDiA 主题向量的分类性能甚至不如直接使用未经主题建模的原始 TF-IDF 向量。这是否意味着在这种情况下 LDiA 毫无用处？先别急于下结论，让我们把主题数量调大，再试一次。

4.5.4 更公平的比较：32 个 LDiA 主题

再来尝试一次，用更多维度、更多主题。也许 LDiA 效率不如 LSA，需要更多主题来分配词汇。尝试 32 个主题（组件）：

```
>>> ldia32 = LDiA(n_components=32, learning_method='batch')
>>> ldia32 = ldia32.fit(bow_docs)
>>> ldia32_topic_vectors = ldia32.transform(bow_docs)
>>> model_ldia32 = LinearDiscriminantAnalysis()
>>> ldia32_performance = evaluate_model(ldia32_topic_vectors,
...     comments.toxic, model_ldia32, 'LDA', 'LDIA (32d)')
>>> hparam_table = hparam_table.append(ldia32_performance,
...     ignore_index=True)
>>> hparam_table.T
```

太好了！将 LDiA 的维度提高后，模型的精确率和召回率几乎都翻倍，F1 分数也大幅提升。更多的主题使 LDiA 对主题划分更精细，至少在这个数据集上，更容易得到线性可分的主题。然而，这些向量表示的性能仍不如 LSA；LSA 能更有效地把评论主题向量分散开来，从而留出更大的间隔供超平面划分类别。

想深入研究，可查看 scikit-learn 和 gensim 中 Dirichlet 分配模型的源码。它们的 API 与 LSA（`sklearn.TruncatedSVD` 和 `gensim.LsiModel`）类似。稍后在摘要任务里还会用到这一点。寻找可解释主题（如本节所示）正是 LDiA 的强项，用这些主题生成分类特征也不难。

快速查找 Python 源码

你之前看到过如何从文档页面浏览 scikit-learn 源码，其实还有更直接的方法——直接在 Python 控制台中查看。任何模块（例如 `sklearn.__file__`）都会暴露其源码路径；在 IPython 中，使用 `??` 即可查看函数、类或对象的源码，例如 `LDA??`：

对于用 C++ 扩展实现的函数或类，此方法无法查看其源码，因为代码被封装在已编译的模块中。

```
>>> import sklearn
>>> sklearn.__file__
'/Users/hobs/anaconda3/envs/conda_env_nlpia/lib/python3.6/site-packages/sklearn/__init__.py'
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>>> LDA??
Init signature: LDA(solver='svd', shrinkage=None, priors=None, n_components=None,
                    store_covariance=False, tol=0.0001)
Source:
class LinearDiscriminantAnalysis(BaseEstimator, LinearClassifierMixin,
                                 TransformerMixin):
    """Linear Discriminant Analysis
    …
```

4.6 距离与相似度

我们需要回到第 2 章和第 3 章讨论的相似度分数，再次确认新的主题向量空间是否适用。同样可以使用相似度（或距离）指标，衡量两个文档在新表示下是多么相似或相距多远。

例如，你可以用相似度分数检验 LSA 主题模型与第 3 章高维 TF-IDF 模型的一致性，看看在剔除大量 BOW 高维信息后，模型在多大程度上保留了这些距离。可以检查主题向量之间的距离是否良好地反映了文档主题之间的关系；理想情况下，主题相似的文档在新向量空间中也应彼此接近。

LSA 可以保留较大的距离，但未必总能保留较小的距离（文档之间的细粒度结构）。其底层 SVD 算法侧重于最大化整个文档集方差，因此可能牺牲某些近距离关系。

特征向量（词向量、主题向量、文档上下文向量等）之间的距离决定了 NLP 流水线或任何机器学习流程的性能。那么，在高维空间中衡量距离有哪些选择？针对具体的 NLP 问题又该选用哪一

种？你也许熟悉一些常见的几何或线性代数示例，但下列许多方法你可能还未接触过：

- **欧几里得/笛卡尔距离（Euclidean/Cartesian distance）或均方根误差（root mean squared error, RMSE）**——2 范数（L2）
- **平方欧几里得距离（Squared Euclidean distance）/平方和距离（sum of squares distance, SSD）**——L2 范数的平方
- **余弦或角距离（Cosine or angular/projected distance）**——归一化点积
- **闵可夫斯基距离（Minkowski distance）**——p 范数或 L_p
- **分式距离、分式范数（Fractional distance, fractional norm）**——$0 < p < 1$ 时的 p 范数或 L_p
- **城市街区、曼哈顿或出租车距离（City block, Manhattan, or taxicab distance）/绝对差总和距离（sum of absolute distance, SAD）**——1 范数（L1）
- **雅卡德距离（Jaccard distance）/集合相似度的倒数（inverse set similarity）**
- **马氏距离（Mahalanobis distance）**
- **莱文斯坦或编辑距离（Levenshtein or edit distance）**

距离计算方法之多足见其重要性。除了 scikit-learn 中成对距离实现外，其他数学领域（如拓扑、统计与工程学）[23] 也广泛使用各类距离。作为参考，下面列出了 `sklearn.metrics` 模块中可用的距离[24]：

代码清单 4.12 scikit-learn 支持的成对距离 Python

```
'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'braycurtis',
'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard',
'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto',
'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
'yule'
```

距离通常可由相似度（similarity）转换而来，反之亦然；距离与相似度成反比。典型的 0—1 归一化转换公式如下：

```
>>> similarity = 1. / (1. + distance)
>>> distance   = (1. / similarity) - 1.
```

对于本身就在 0—1 范围（概率）内的度量，更常见的转换是：

```
>>> similarity = 1. - distance
>>> distance   = 1. - similarity
```

余弦距离有自己的取值范围惯例：两向量的角距离通常表示为二者夹角相对于最大可能夹角（180° 或 π 弧度）[25] 的比例：

```
>>> import math
>>> angular_distance = math.acos(cosine_similarity) / math.pi
>>> distance = 1. / similarity - 1.
>>> similarity = 1. - distance
```

我们为何花如此多篇幅讨论距离？因为在本书最后一节将介绍**语义搜索**。其思想是找出与查询在语义上最相近（或最远）的文档；在示例应用中将使用余弦相似度，但如你所见，还有多种度量文档相似度的方法。

4.7 使用反馈进行引导（Steering with feedback）

此前的语义分析方法都未考虑文档之间的相似度信息。我们创建的主题向量最优于一组通用规则；在无监督学习特征（主题）提取时，并没有任何关于主题向量应彼此靠近程度的“反馈”。

可引导或学习得到的距离度量（steered, or learned distance metrics）[26] 是降维与特征提取的最新进展。通过在聚类与嵌入算法中调整距离分数，你可以“引导”向量，使其最小化某个代价函数，从而迫使向量聚焦于你感兴趣的信息。

在前述 LSA 章节里，你忽略了文档的元信息——比如在评论案例中，你忽略了评论作者。这其实是主题相似度的重要指示器，可用于改进 LSA 的向量转换。

某猎头公司在实践中尝试用简历与职位描述的主题向量的余弦距离来做匹配，效果还行，但很快发现：若根据候选人与客户经理的反馈来“引导”主题向量，会得到更佳结果。好匹配的简历—职位向量被拉得更近，其余配对则被拉远。

一种做法是计算两类质心（如 LDA 中所做）之间的平均差异，并把其中一部分“偏置”加到所有简历或职位向量上，以抵消总体向量差异。这样可去除简历与职位描述间的平均主题差别。诸如 *beer on tap at lunch* 这类仅在职位描述中出现、而简历里从未出现的主题，以及 *underwater sculpture* 等仅在少数简历中出现、但职位描述中从未出现的主题都会被削弱，引导后的主题向量更专注于真正想要建模的主题。

4.8 主题向量的威力（Topic vector power）

利用主题向量，你可以比较单词、文档、语句、语料库的含义；发现相似文档与语句的簇群；不再仅靠词频或词汇，而是真正按语义相关性检索文档。

这被称为**语义搜索（semantic search）**，不要与 **语义网（semantic web）** 混淆[27]。语义搜索正是先进搜索引擎的强项：即使查询中的词语很少出现在目标文档里，也能找到与你需求最匹配的结果。例如，它能区分 *The Cheese Shop* 中的 *Python* 软件包与佛罗里达水族馆里的蟒蛇，同时还能识别两者与 *Ruby gem* 的相似度[28]。

语义搜索为寻找与生成有意义文本提供了重要工具，但人脑不擅长处理三维以上的高维对象、向量、超平面与超立方体。我们的直觉在三维以上便会失效。例如，使用二维向量（经纬度坐标）在 Google 地图上搜索附近咖啡店非常直观；但若要在高维向量空间（超空间）中划分超平面与超立方体作为搜索边界，往往既不实际，也常常不可能。

正如 Geoffrey Hinton 所说："要处理 14 维空间的超平面，试着想象一个 3D 空间，然后对自己默念 '14'。"[29] 就像在《Flatland》中，你可以借由 2D 可视化来稍微窥见 3D 的影子。本章多次使用 2D 可视化帮助你探索 3D 世界中的 4D 阴影。在进入下一节的语义搜索前，不妨回看图 4.1 的 3D 主题向量，想象如果再多一个主题、创建 4D 的语言意义空间会是什么样。若此时你还不头疼，说明你还未真正思考 4D 主题向量；当你感到大脑"燃烧"时，别忘了：维度从 2D 到 3D 的增长只是线性，而从 3D 到 4D 的增长却是指数级——这就是"维度诅咒"。

4.8.1 语义搜索（Semantic search）

当你根据文档中包含的某个词或词的一部分来检索文档时，这被称为**全文检索**（full-text search）。搜索引擎正是这样做的：它们将文档切分成可以通过**倒排索引**（inverted index）检索的片段（通常是词语），就像你在教材索引页看到的那样。这样做需要大量的记录工作和猜测来处理拼写错误与打字错误，但效果相当不错。

语义搜索是在全文检索的基础上，引入查询词与目标文档"含义"这一维度的检索方式。本章你学习了两种方法——LSA 和 LDiA——用固定长度的向量来表示词语和文档的语义（含义），与数据集大小和内容无关。最早被称作"潜在语义索引"（latent semantic **indexing**）的潜在语义分析之所以命名为索引，是因为当时希望用数值索引（如 BOW 和 TF—IDF 表）实现语义检索。随着更好的向量表达方式出现（后续章节会介绍），语义搜索变得越来越重要。伴随生成式模型的爆炸式增长以及**检索增强生成**（retrieval-augmented generation, RAG）技术的普及，许多公司开始存储并检索表示文本含义的向量。Knowt 是一个开源项目，旨在创建私有语义搜索向量数据库，并将 RAG 虚拟助手应用于私有自然语言文档，例如个人日记或医疗记录。[30, 31]

但与 BOW 或 TF—IDF 表不同，**语义向量表**无法使用传统倒排索引技术[32]轻松索引。倒排索引适用于离散或二值向量，因为它只需为每个非零离散维度维护一个条目；而 TF—IDF 向量稀疏、零值居多，你无需为大多数文档的大多数维度建索引。[33]

LSA 与 LDiA 产生的主题向量既高维、连续，又稠密（几乎没有零）。语义分析算法并不能生成可扩展搜索的高效索引；事实上，上节讨论的"维度诅咒"使得精确索引几乎不可能。"潜在语义索

引”中的 *indexing* 更像是一种美好愿景而非现实，亦因此 LSA 逐渐成为描述生成主题向量的语义分析算法的更流行术语。

应对高维向量挑战的一个方案是使用**局部敏感哈希**（locality-sensitive hash, LSH）。LSH 像邮政编码一样，为超空间中的某一区域指定哈希，以便日后快速定位；与普通哈希一样，它是确定性的，只依赖向量中的值。但当维度超过 12 时，LSH 的效果就不尽如人意了。图 4–5 中，每一行代表不同的向量维度（从 2D 到 16D），使用的是你在短信垃圾过滤问题中用过的向量。该表展示了如果使用 LSH 索引大量语义向量，搜索结果的准确率会如何变化。

Dimensions	100th cosine distance	Top 1 correct	Top 2 correct	Top 10 correct	Top 100 correct
2	.00	TRUE	TRUE	TRUE	TRUE
3	.00	TRUE	TRUE	TRUE	TRUE
4	.00	TRUE	TRUE	TRUE	TRUE
5	.01	TRUE	TRUE	TRUE	TRUE
6	.02	TRUE	TRUE	TRUE	TRUE
7	.02	TRUE	TRUE	TRUE	FALSE
8	.03	TRUE	TRUE	TRUE	FALSE
9	.04	TRUE	TRUE	TRUE	FALSE
10	.05	TRUE	TRUE	FALSE	FALSE
11	.07	TRUE	TRUE	TRUE	FALSE
12	.06	TRUE	TRUE	FALSE	FALSE
13	.09	TRUE	TRUE	FALSE	FALSE
14	.14	TRUE	FALSE	FALSE	FALSE
15	.14	TRUE	TRUE	FALSE	FALSE
16	.09	TRUE	TRUE	FALSE	FALSE

图 4–5　语义搜索准确率在约 12 维时开始下降

当维度超过 16 时，你很难返回两条真正有用的搜索结果。

要在 100 维向量上执行语义搜索且没有索引怎么办？你已经知道如何用 LSA 将查询字符串转换为主题向量，也知道如何用余弦相似度（标量积、内积或点积）比较两个向量的相似度以找到最相近的匹配。若要获得精确的语义匹配，需要找到与查询（搜索）主题向量最近的所有文档主题向量（专业术语称为**穷举搜索**）。但如果有 *n* 个文档，就得对查询向量做 *n* 次比较，这需要大量的点积运算。

你可以用 NumPy 将运算向量化成矩阵乘法，但这并不会减少运算次数——只会让它们快上 100 倍[34]。从根本上说，精确语义搜索仍需要对每次查询执行 O(N) 次乘加运算，因此其扩展能力仅与语料规模线性相关，这在大规模语料（如 Google 搜索或 Wikipedia 语义搜索）上难以奏效。

关键是**追求“够好”而非完美索引**或 LSH。现有多个高效且准确的**近似最近邻**（approximate nearest neighbors）算法实现了利用 LSH 高效执行语义搜索（第 10 章将详细讨论）。严格来说，这些索引或哈希解决方案无法保证找出与你的语义搜索查询最匹配的全部结果，但能在几乎与传统倒排索引同样快的速度下，提供一组足够接近的匹配结果，只要你愿意牺牲一点精度。[35]

4.9 为你的机器人装备语义搜索

让我们用你在主题建模中学到的新知识，改进上一章开始构建的机器人。本节聚焦同一任务：**问答系统**。整体思路与第 3 章的代码非常相似：仍然使用向量表示来找到数据集中与查询最相似的问题，但这一次，我们使用更能够表达含义的语义表示。

首先，像上一章一样加载问答数据：

```
>>> REPO_URL = 'https://gitlab.com/tangibleai/community/qary-cli/-/raw/main'
>>> FAQ_DIR  = 'src/qary/data/faq'
>>> FAQ_FILENAME = 'short-faqs.csv'
>>> DS_FAQ_URL = '/'.join([REPO_URL, FAQ_DIR, FAQ_FILENAME])

>>> df = pd.read_csv(DS_FAQ_URL)
```

下一步是把所有问题与查询都表示成向量。由于数据集较小，无需应用 LSH 或其他索引算法；逐一遍历问题并选择最佳匹配即可。

代码清单 4.13　为机器人创建基于语义相似度的回答函数　Python

```
>>> vectorizer = TfidfVectorizer()
>>> vectorizer.fit(df['question'])
>>> tfidfvectors = vectorizer.transform(df['question'])
>>> svd = TruncatedSVD(n_components=16, n_iter=100)
>>> tfidfvectors_16d = svd.fit_transform(tfidfvectors)

>>> def bot_reply(question):
...     question_tfidf = vectorizer.transform([question]).todense()
...     question_16d = svd.transform(np.asarray(question_tfidf))
...     idx = question_16d.dot(tfidfvectors_16d.T).argmax()
...     print(
...         f"Your question:\n {question}\n\n"
...         f"Most similar FAQ question:\n {df['question'][idx]}\n\n"
...         f"Answer to that FAQ question:\n {df['answer'][idx]}\n\n"
...     )
```

简单测试

```
>>> bot_reply("What's overfitting a model?")
Your question:
 What's overfitting a model?
Most similar FAQ question:
 What is overfitting?
Answer to that FAQ question:
```

（此处显示对应答案）

更难的问题

```
>>> bot_reply("How do I decrease overfitting for Logistic Regression?")
Your question:
 How do I decrease overfitting for Logistic Regression?
Most similar FAQ question:
 How to reduce overfitting and improve test set accuracy for a
   ➥ LogisticRegression model?
Answer to that FAQ question:
 Decrease the C value, this increases the regularization strength.
```

看来新版机器人不仅识别了 *decrease* 与 *reduce* 的同义关系，还理解了 *Logistic Regression* 与 *LogisticRegression* 几乎相同——TF-IDF 模型几乎无法做到这一点。你离构建真正强大的问答系统又近了一步。

4.10 自测

1. 为了让 LDiA 更高效地进行主题建模，你会使用哪些预处理技术？对于 LSA 又会如何？
2. 能想到一个数据集或问题，使得 TF-IDF 的表现优于 LSA 吗？反之亦然呢？
3. 我们提到对 LDiA 进行停用词过滤作为预处理步骤。在哪些情况下过滤停用词会有益？
4. 语义搜索面临的主要挑战是 LSA 的稠密主题向量无法用倒排索引检索。你能解释原因吗？

本章小结

- 通过分析数据集中词项的共现，你可以推导出词语与文档的含义。
- SVD 可用于语义分析，将 TF-IDF 与 BOW 向量分解并转换为主题向量。
- 超参数表可用于比较不同流水线与模型的性能。

- 当需要可解释的主题分析时，可使用 LDiA。
- 无论你如何创建主题向量，都可以用语义搜索根据含义检索文档。

本章注释

1 本章在讨论主题分析时使用 **主题向量（topic vector）** 一词，而第 6 章在介绍 Word2Vec 时使用 **词向量（word vector）**。正式的 NLP 书籍（如 Jurafsky 和 Martin 的 *The NLP Bible*，https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf#chapter.15）采用 topic vector；而 *Semantic Vector Encoding and Similarity Search*（https://arxiv.org/pdf/1706.00957.pdf）等资料使用*semantic vector*。

2 词干提取（stemming）与词形还原（lemmatization）都会删除或修改词尾和词前缀，即单词最后几个字符。若要识别拼写相近（或拼错）的单词，编辑距离（edit-distance）计算更合适。

3 我们非常喜欢用 Google 的 Ngram Viewer 来可视化这种趋势：https://mng.bz/ZoyA。

4 斯坦福的 Doug Lenat 正在尝试把常识编码进算法。可参见 *Wired* 杂志文章“One Genius' Lonely Crusade to Teach a Computer Common Sense”（https://www.wired.com/2016/03/doug-lenat-artificial-intelligence-common-sense-engine）。

5 **词素（morpheme）** 是构成单词的最小有意义部分。参见维基百科“Morpheme”条目（https://en.wikipedia.org/wiki/Morpheme）。

6 维基百科“Topic Model”页面有一段视频展示 LSA 的直观原理（https://upload.wikimedia.org/wikipedia/commons/7/70/Topic_model_scheme.webm#t=00:00:01,00:00:17.600）。

7 该数据集的更大版本曾作为 2017 年 Kaggle 竞赛（https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge）的基础，并由 Jigsaw 以 CC0 许可证公开。

8 聚类的 **质心（centroid）** 是指该簇内所有点坐标平均值所对应的点。

9 若想进一步直观理解 **精准率（precision）** 和 **召回率（recall）**，请参阅维基百科相关文章（https://en.wikipedia.org/wiki/Precision_and_recall），里面有很好的可视化示例。

10 关于在某些情况下**不使用 F1 分数（F1 score）**的理由及其他评价指标，可参阅“F-Score”维基百科条目（https://en.wikipedia.org/wiki/F-score）。

11 scikit-learn 文档中提供了两种估计器的可视化示例：https://scikit-learn.org/dev/modules/lda_qda.html。

12 想更深入了解降维，可阅读 Hussein Abdullaif 的四篇系列博文：http://mng.bz/RIRv。

13 实际上执行主成分分析（PCA）主要有两种方法；可查看维基百科“Principal Component Analysis”条目（https://en.wikipedia.org/wiki/Principal_component_analysis#Singular_value_decomposition），了解另一种方法及其与本书所述方法几乎得出相同结果的原因。

14 若想了解 **全 SVD（full SVD）** 及其其他应用，可阅读维基百科“Singular Value Decomposition”条目：https://en.wikipedia.org/wiki/Singular_value_decomposition。

15 参见维基百科“Overfitting”条目（https://en.wikipedia.org/wiki/Overfitting）。

16 点击 scikit-learn 文档左上角的 *Source* 链接即可查看任何函数源码。

17 想深入了解 PCA 的数学原理，请访问：https://en.wikipedia.org/wiki/Principal_component_analysis。

18 Sonia Bergamaschi 与 Laura Po 2015 年的一项内容推荐算法比较显示，LSA 的准确率约为 LDiA 的两倍。参见“Comparing LDA and LSA Topic Models for Content-Based Movie Recommendation Systems”（https://dbgroup.ing.unimore.it/~po/pubs/LNBI_2015.pdf）。

19 参见“Latent Dirichlet Allocation,” David M. Blei、Andrew Y. Ng 与 Michael I. Jordan（http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf）。

20 想了解 LDiA 目标函数的细节，可阅读原始论文“Online Learning for Latent Dirichlet Allocation”（Matthew D. Hoffman, David M. Blei, Francis Bach, https://www.di.ens.fr/%7Efbach/mdhnips2010.pdf）。

21 Blei 与 Ng 使用希腊字母 θ 而不是 **k** 来表示该参数。

22 若想了解为何过拟合有害以及**泛化（generalization）**的作用，请参见附录 D。

23 更多距离度量可参考 Math.NET Numerics（https://numerics.mathdotnet.com/Distance.html）。

24 scikit-learn 中 `sklearn.metrics` 模块的距离度量文档见：https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html。

25 参见维基百科“Cosine Similarity”条目（https://en.wikipedia.org/wiki/Cosine_similarity）。

26 参见“Eccv Sggraph”论文（http://users.cecs.anu.edu.au/~sgould/papers/eccv14-sggraph.pdf）。

27 **语义网（semantic web）** 是在 HTML 文档中使用标签结构化自然语言文本，以便标签层级及其内容提供元素（文本、图片、视频等）之间关系（连接网络）信息的做法。

28 Ruby 是一门编程语言，其软件包称为 **gems**。

29 参见 https://www.cs.toronto.edu/~hinton/coursera/lecture2/lec2c.mp4。

30 你可从 GitLab 源码仓库下载安装 **knowt**（https://gitlab.com/tangibleai/community/knowt/）。

31 如果只想在自己的数据上使用，可以 `pip install knowt`（https://pypi.org/project/knowt/）。

32 高维数据聚类等同于用边界框离散化或索引高维数据，维基百科“Clustering_high-dimensional_data”条目对此有描述（https://en.wikipedia.org/wiki/Clustering_high-dimensional_data）。

33 参见维基百科“Inverted Index”条目（https://en.wikipedia.org/wiki/Inverted_index）。

34 将 Python 代码向量化（尤其是把成对距离计算的双层 for 循环改写为向量化操作）可使代码速度提升近 100 倍。参见 Hacker Noon 文章“Vectorizing the Loops with NumPy”（https://hackernoon.com/speeding-up-your-code-2-vectorizing-the-loops-with-numpy-e380e939bed3）。

35 若想了解更快速查找高维向量最近邻的方法，可参阅第 10 章，或直接使用 Spotify 的 **annoy** 包为主题向量建立索引。