AI基础：特征工程-文本特征处理

原创：机器学习初学者 机器学习初学者 1周前

0.导语

特征工程到底是什么呢？顾名思义，其本质是一项工程活动，目的是最大限度地从原始数据中提取特征以供算法和模型使用。

在此之前，我已经写了以下几篇AI基础的快速入门，本篇文章讲解特征工程基础第三部分：（文本特征处理）。

目前已经发布：

AI 基础：Python 简易入门

AI 基础：Numpy 简易入门

AI 基础：Pandas 简易入门

AI 基础：Scipy(科学计算库) 简易入门

AI基础：数据可视化简易入门（matplotlib和seaborn）

AI基础：特征工程-类别特征

AI基础：特征工程-数字特征处理

后续持续更新

参考资料：

[1]原版（英文）图书地址:
https://www.oreilly.com/library/view/feature-engineering-for/9781491953235/
[2]翻译来源apachecn:
https://github.com/apachecn
[3]翻译作者@kkejili:
https://github.com/kkejili

代码修改和整理：黄海广，原文修改成jupyter notebook格式，并增加和修改了部分代码，测试全部通过，所有数据集已经放在百度云下载。

本文代码可以在**github**下载：

https://github.com/fengdu78/Data-Science-Notes/tree/master/9.feature-engineering

数据集的百度云：

链接：**https://pan.baidu.com/s/1uDXt5jWUOfI0fS7hD91vBQ** 提取码：**8p5d**

三、文本数据: 展开、过滤和分块

如果让你来设计一个算法来分析以下段落，你会怎么做？

```
Emma knocked on the door. No answer. She knocked again and waited. There was a large maple tree nex
```

该段包含很多信息。我们知道它谈到了到一个名叫Emma的人和一只乌鸦。这里有一座房子和一棵树，艾玛正想进屋，却看到了乌鸦。这只华丽的乌鸦注意到艾玛，她有点害怕，但正在尝试交流。

那么，这些信息的哪些部分是我们应该提取的显着特征？首先，提取主要角色艾玛和乌鸦的名字似乎是个好主意。接下来，注意房子，门和树的布置可能也很好。关于乌鸦的描述呢？Emma的行为呢，敲门，退后一步，打招呼呢？

本章介绍文本特征工程的基础知识。我们从词袋（bags of words）开始，这是基于字数统计的最简单的文本功能。一个非常相关的变换是 tf-idf，它本质上是一种特征缩放技术。它将被我在（下一篇）章节进行全面讨论。本章首先讨论文本特征提取，然后讨论如何过滤和清洗这些特征。

Bag of X：把自然文本变成平面向量

无论是构建机器学习模型还是特征工程，其结果应该是通俗易懂的。简单的事情很容易尝试，可解释的特征和模型相比于复杂的更易于调试。简单和可解释的功能并不总是会得到最精确的模型。但从简单开始就是一个好主意，仅在绝对必要时我们可以增加其复杂性。

对于文本数据，我们可以从称为 BOW 的字数统计开始。字数统计表中并没有特别费力来寻找 `"Emma"` 或乌鸦这样有趣的实体。但是这两个词在该段落中被重复提到，并且它们在这里的计数比诸如 `"hello"` 之类的随机词更高。对于此类简单的文档分类任务，字数统计通常比较适用。它也可用于信息检索，其目标是检索与输入文本相关的文档集。这两个任务都很好解释词级特征，因为某些特定词的存在可能是本文档主题内容的重要指标。

词袋

在词袋特征中，文本文档被转换成向量。（向量只是 n 个数字的集合。）向量包含词汇表中每个单词可能出现的数目。如果单词 `"aardvark"` 在文档中出现三次，则该特征向量在与该单词对应的位置上的计数为 3。如果词汇表中的单词没有出现在文档中，则计数为零。例如，“这是一只小狗，它是非常可爱”的句子具有如图所示的 BOW 表示

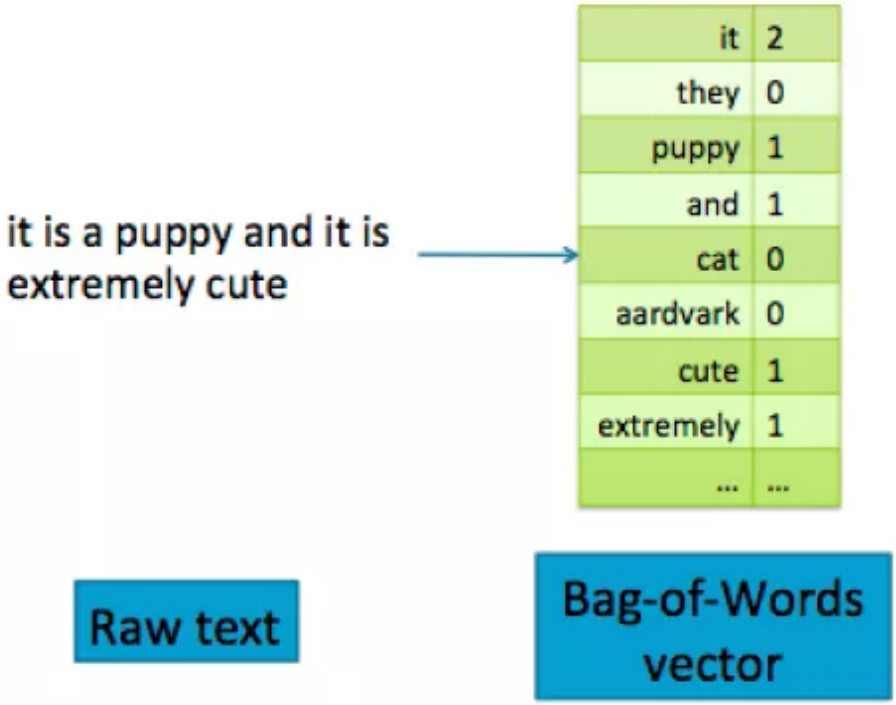


图 3-1 转换词成向量描述图

BOW 将文本文档转换为平面向量。它是“平面的”，因为它不包含任何原始的文本结构。原文是一系列词语。但是词袋向量并没有序列；它只是记得每个单词在文本中出现多少次。它不代表任何词层次结构的概念。例如，“动物”的概念包括“狗”，“猫”，“乌鸦”等。但是在一个词袋表示中，这些词都是矢量的相同元素。

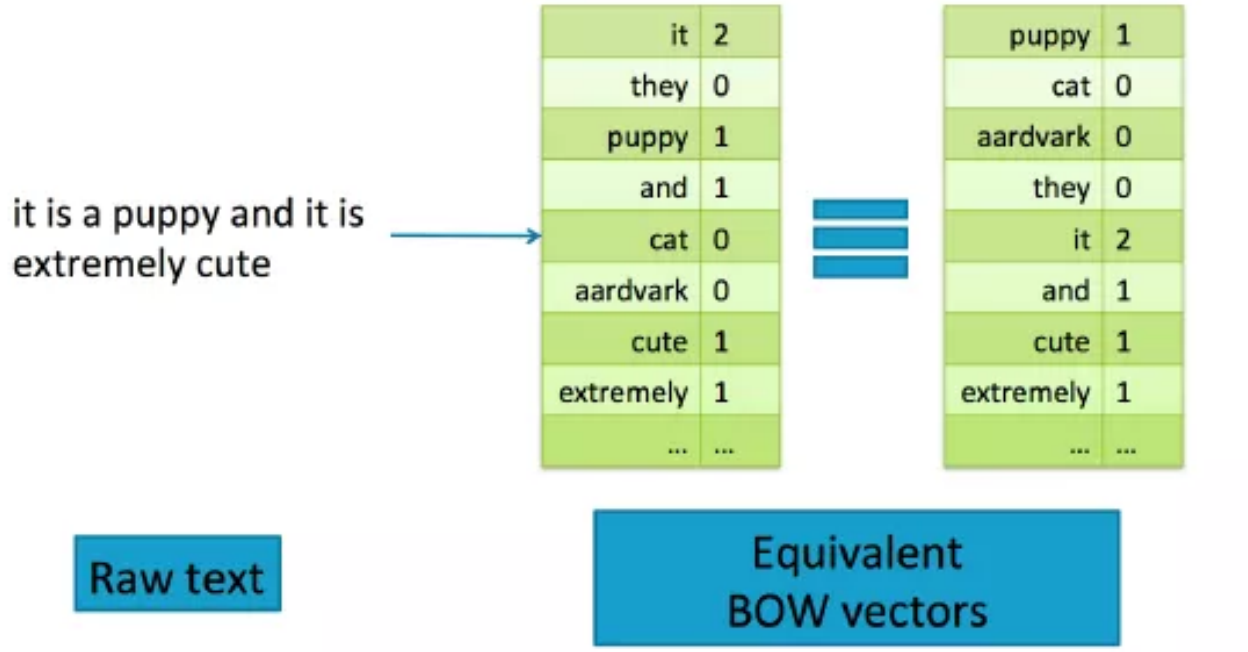


图 3-2 两个等效的词向量，向量中单词的排序不重要，只要它在数据集中的个数和文档中出现数量是一致的。

重要的是特征空间中数据的几何形状。在一个词袋矢量中，每个单词成为矢量的一个维度。如果词汇表中有 n 个单词，则文档将成为n维空间中的一个点。很难想象二维或三维以外的任何物体的几何形状，所以我们必须使用我们的想象力。图3-3显示了我们的例句在对应于“小狗”和“可爱”两个维度的特征空间中的样子。

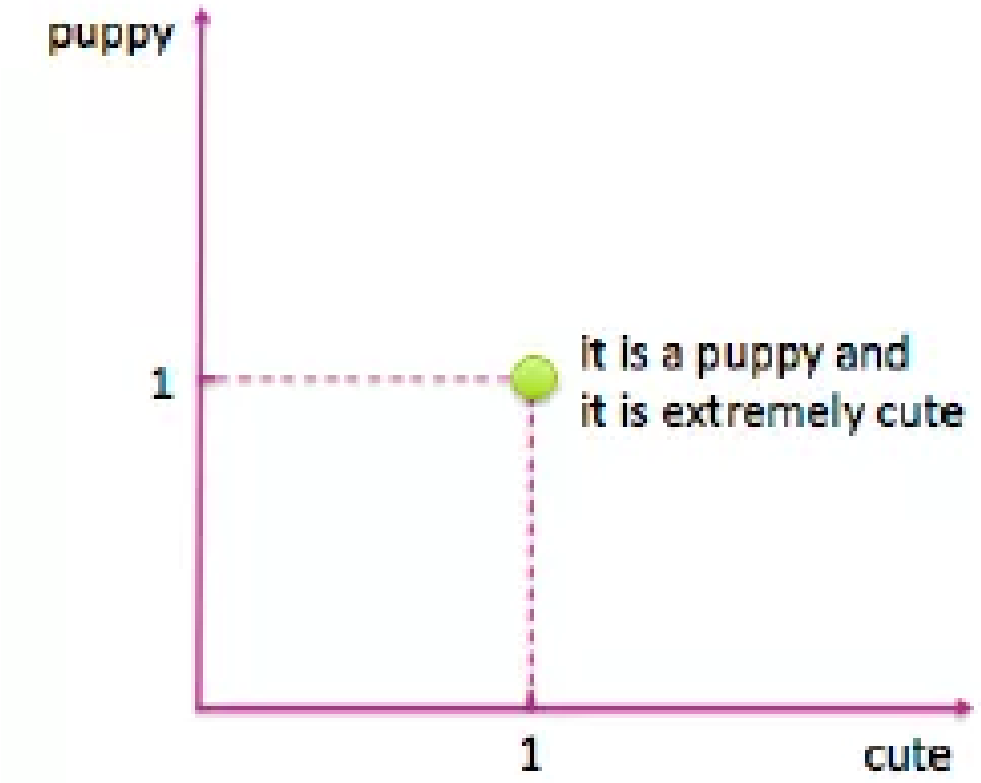


图 3-3 特征空间中文本文档的图示

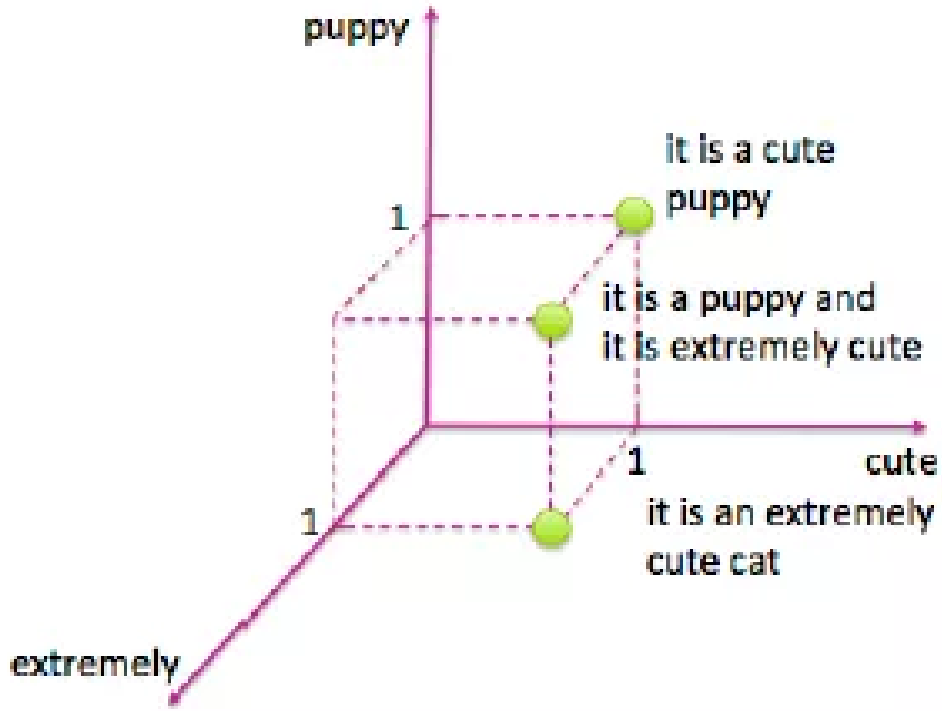
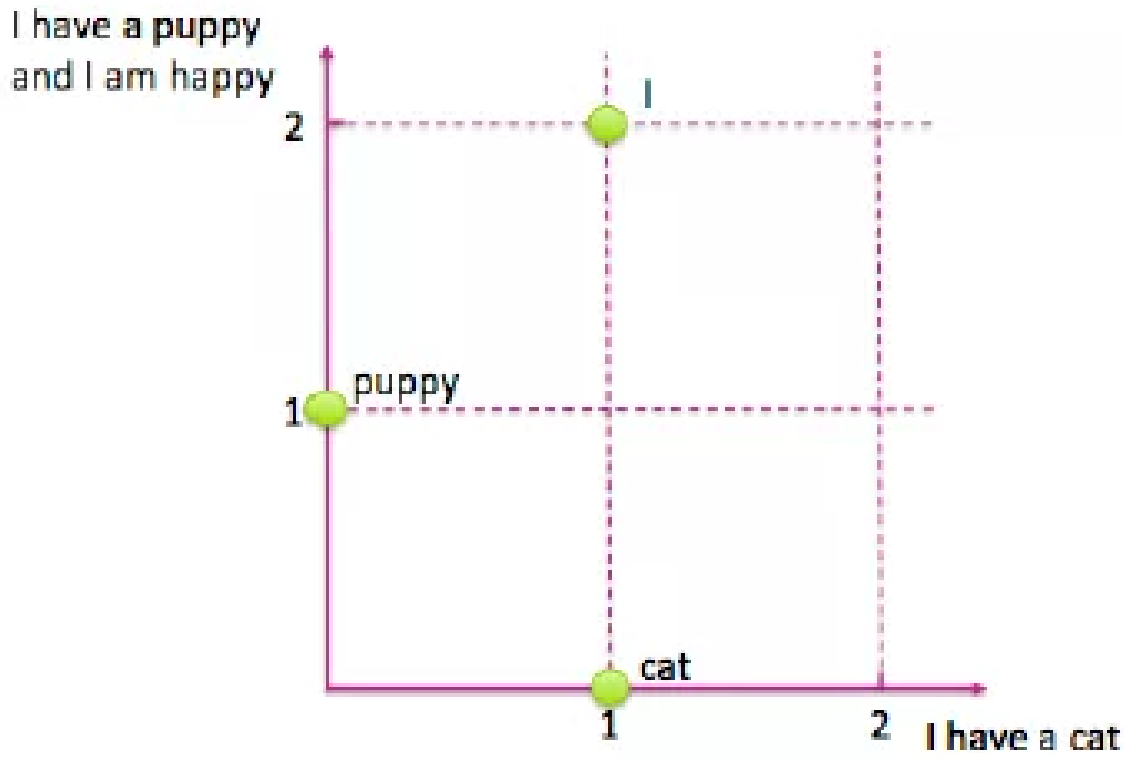


图 3-4 三维特征空间

图 3-3 和图 3-4 描绘了特征空间中的数据向量。坐标轴表示单个单词，它们是词袋表示下的特征，空间中的点表示数据点（文本文档）。有时在数据空间中查看特征向量也是有益的。特征向量包含每个数据点中特征的值。轴表示单个数据点和点表示特征向量。图 3-5 展示了一个例子。通过对文本文档进行词袋特征化，一个特征是一个词，一个特征向量包含每个文档中这个

词的计数。这样，一个单词被表示为一个“一个词向量”。正如我们将在第 4 章中看到的那样，这些文档词向量来自词袋向量的转置矩阵。



Bag-of-N-gram

Bag-of-N-gram 或者 bag-of-ngram 是 BOW 的自然延伸。n-gram 是 n 个有序的记号（token）。一个词基本上是一个 1-gram，也被称为一元模型。当它被标记后，计数机制可以将单个词进行计数，或将重叠序列计数为 n-gram。例如，`"Emma knocked on the door"` 这句话会产生 n-gram，如 `"Emma knocked"`，`"knocked on"`，`"on the"`，`"the door"`。N-gram 保留了文本的更多原始序列结构，故 bag-of-ngram可以提供更多信息。但是，这是有代价的。理论上，用 k 个独特的词，可能有 k 个独立的 2-gram（也称为 bigram）。在实践中，并不是那么多，因为不是每个单词后都可以跟一个单词。尽管如此，通常有更多不同的 n-gram（n > 1）比单词更多。这意味着词袋会更大并且有稀疏的特征空间。这也意味着 n-gram 计算，存储和建模的成本会变高。n 越大，信息越丰富，成本越高。

为了说明随着 n 增加 n-gram 的数量如何增加，我们来计算纽约时报文章数据集上的 n-gram。我们使用 Pandas 和 scikit-learn 中的 `CountVectorizer` 转换器来计算前 10,000 条评论的 n-gram。

```
import pandas as pd
import json
```

```
from sklearn.feature_extraction.text import CountVectorizer
# Load the first 10,000 reviews
f = open('data/yelp_academic_dataset_review.json')
js = []
for i in range(10000):
    js.append(json.loads(f.readline()))
f.close()
review_df = pd.DataFrame(js)
```

备注：所有数据集已经放在百度云下载。

```
review_df.head()
```

	business_id	date	review_id	stars	text	type	user_id	votes
0	9yKzy9PApeiPPOUJEtnvkg	2011-01-26	fWKvX83p0-ka4JS3dc6E5A	5	My wife took me here on my birthday for breakf...	review	rLtl8ZkDX5vH5nAx9C3q5Q	{'funny': 0, 'useful': 5, 'cool': 2}
1	ZRJwVLyzEJq1VAihDhYiow	2011-07-27	IjZ33sJrzXqU-0X6U8NwyA	5	I have no idea why some people give bad review...	review	0a2KyEL0d3Yb1V6aivbIuQ	{'funny': 0, 'useful': 0, 'cool': 0}
2	6oRAC4uyJCsJl1X0WZpVSA	2012-06-14	IESLBzqUCLdSzSqm0eCSxQ	4	love the gyro plate. Rice is so good and I als...	review	0hT2KtfLiobPvh6cDC8JQg	{'funny': 0, 'useful': 1, 'cool': 0}

	business_id	date	review_id	stars	text	type	user_id	votes
3	_1QQZuf4zZOyFCvXc0o6Vg	2010-05-27	G-WvGaISbqqaMHlNnByodA	5	Rosie, Dakota, and I LOVE Chaparral Dog Park!!...	review	uZetl9T0NcROGOyFfughhg	{'funny': 0, 'useful': 2, 'cool': 1}
4	6ozycU1RpktNG2-1BroVtw	2012-01-05	1uJFq2r5QfJG_6ExMRCaGw	5	General Manager Scott Petello is a good egg!!!...	review	vYmM4KTsC8ZfQBg-j5MWkw	{'funny': 0, 'useful': 0, 'cool': 0}

```
# Create feature transformers for unigram, bigram, and trigram.
# The default ignores single-character words, which is useful in practice because it trims
# uninformative words. But we explicitly include them in this example for illustration purposes.
bow_converter = CountVectorizer(token_pattern='(?u)\\b\\w+\\b')
bigram_converter = CountVectorizer(
    ngram_range=(2, 2), token_pattern='(?u)\\b\\w+\\b')
trigram_converter = CountVectorizer(
    ngram_range=(3, 3), token_pattern='(?u)\\b\\w+\\b')
# Fit the transformers and look at vocabulary size
bow_converter.fit(review_df['text'])
words = bow_converter.get_feature_names()
bigram_converter.fit(review_df['text'])
bigram = bigram_converter.get_feature_names()
trigram_converter.fit(review_df['text'])
trigram = trigram_converter.get_feature_names()
print(len(words), len(bigram), len(trigram))
```

```
29222 368943 881620
```

```
# Sneak a peek at the ngram themselves
words[:10]
```

```
['0', '00', '000', '007', '00a', '00am', '00pm', '01', '02', '03']
```

```
bigram[-10:]
```

```
['zuzu was',
 'zuzus room',
 'zweigel wine',
 'zwiebel kräuter',
 'zy world',
 'zzed in',
 'éclairs napoleons',
 'école lenôtre',
 'ém all',
 'òc châm']
```

```
trigram[:10]
```

```
['0 0 eye',
 '0 20 less',
 '0 39 oz',
 '0 39 pizza',
 '0 5 i',
 '0 50 to',
 '0 6 can',
 '0 75 oysters',
 '0 75 that',
 '0 75 to']
```

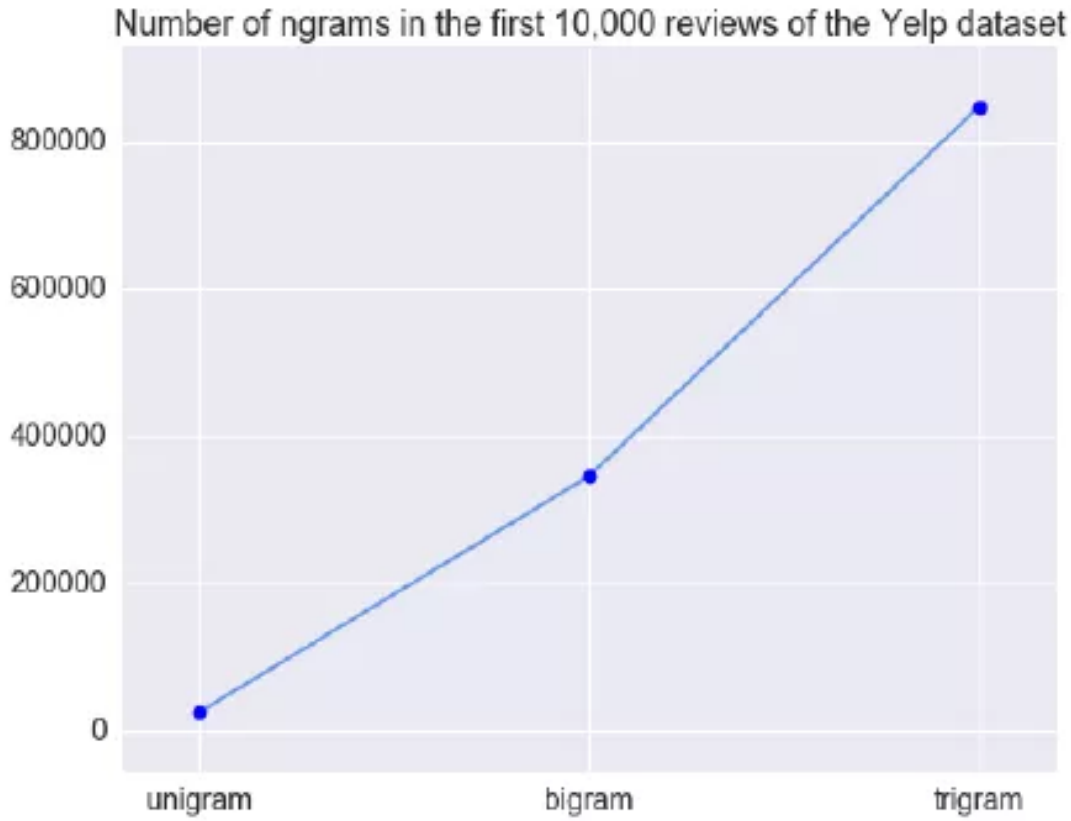


图3-6 Number of unique n-gram in the first 10,000 reviews of the Yelp dataset

过滤清洗特征

我们如何清晰地将信号从噪声中分离出来？通过过滤，使用原始标记化和计数来生成简单词表或 n-gram 列表的技术变得更加可用。短语检测，我们将在下面讨论，可以看作是一个特别的 bigram 过滤器。以下是执行过滤的几种方法。

停用词

分类和检索通常不需要对文本有深入的理解。例如，在 `"Emma knocked on the door"` 一句中， `"on"` 和 `"the"` 这两个词没有包含很多信息。代词、冠词和介词大部分时间并没有显示出其价值。流行的 Python NLP 软件包 NLTK 包含许多语言的语言学家定义的停用词列表。（您将需要安装 NLTK 并运行 `nltk.download()` 来获取所有的好东西。）各种停用词列表也可以在网上找到。例如，这里有一些来自英语停用词的示例词

```
Sample words from the nltk stopword list
a, about, above, am, an, been, didn’t, couldn’t, i’d, i’ll, itself, let’s, myself, our, they, throu
```

请注意，该列表包含撇号，并且这些单词没有大写。为了按原样使用它，标记化过程不得去掉撇号，并且这些词需要转换为小写。

基于频率的过滤

停用词表是一种去除空洞特征常用词的方法。还有其他更统计的方法来理解“常用词”的概念。在搭配提取中，我们看到依赖于手动定义的方法，以及使用统计的方法。同样的想法也适用于文字过滤。我们也可以使用频率统计。

高频词

频率统计对滤除语料库专用常用词以及通用停用词很有用。例如，纽约时报文章数据集中经常出现“纽约时报”和其中单个单词。“议院”这个词经常出现在加拿大议会辩论的Hansard语料库中的“众议院”一词中，这是一种用于统计机器翻译的流行数据集，因为它包含所有文档的英文和法文版本。这些词在普通语言中有意义，但不在语料库中。手动定义的停用词列表将捕获一般停用词，但不是语料库特定的停用词。

表 3-1 列出了 Yelp 评论数据集中最常用的 40 个单词。在这里，频率被认为是它们出现在文件（评论）中的数量，而不是它们在文件中的数量。正如我们所看到的，该列表涵盖了许多停用

词。它也包含一些惊喜。 `"s"` 和 `"t"` 在列表中，因为我们使用撇号作为标记化分隔符，并且诸如 `"Mary's"` 或 `"did not"` 之类的词被解析为 `"Mary s"` 和 `"didn t"` 。

词 `"good"` ， `"food"` 和 `"great"` 分别出现在三分之一的评论中。但我们可能希望保留它们，因为它们对于情感分析或业务分类非常有用。

Table 3-1. Most frequent words in the Yelp reviews dataset

Rank	Word	Document Frequency	Rank	Word	Document Frequency
1	the	1416058	21	t	684049
2	and	1381324	22	not	649824
3	a	1263126	23	s	626764
4	i	1230214	24	had	620284
5	to	1196238	25	so	608061
6	it	1027835	26	place	601918
7	of	1025638	27	good	598393
8	for	993430	28	at	596317
9	is	988547	29	are	585548
10	in	961518	30	food	562332
11	was	929703	31	be	543588
12	this	844824	32	we	537133
13	but	822313	33	great	520634
14	my	786595	34	were	516685
15	that	777045	35	there	510897
16	with	775044	36	here	481542
17	on	735419	37	all	478490
18	they	720994	38	if	475175
19	you	701015	39	very	460796
20	have	692749	40	out	460452

最常用的单词最可以揭示问题，并突出显示通常有用的单词通常在该语料库中曾出现过多次。例如，纽约时报语料库中最常见的词是“时代”。实际上，它有助于将基于频率的过滤与停用词列表结合起来。还有一个棘手的问题，即何处放置截止点。不幸的是这里没有统一的答案。在大多数情况下截断还需手动确定，并且在数据集改变时可能需要重新检查。

稀有词

根据任务的不同，可能还需要筛选出稀有词。对于统计模型而言，仅出现在一个或两个文档中的单词更像噪声而非有用信息。例如，假设任务是根据他们的 Yelp 评论对企业进行分类，并且单个评论包含 `"gobbledygook"` 这个词。基于这一个词，我们将如何说明这家企业是餐厅，美容院还是一间酒吧？即使我们知道在这种情况下的这种生意发生在酒吧，它也会对于其他包含 `"gobbledygook"` 这个词的评论来说，这可能是一个错误。

不仅稀有词不可靠，而且还会产生计算开销。这套 160 万个 Yelp 评论包含 357,481 个独特单词（用空格和标点符号表示），其中 189,915 只出现在一次评论中，41,162 次出现在两次评论中。超过 60% 的词汇很少发生。这是一种所谓的重尾分布，在现实世界的数据中非常普遍。许多统计机器学习模型的训练时间随着特征数量线性地变化，并且一些模型是二次的或更差的。稀有词汇会产生大量的计算和存储成本，而不会带来额外的收益。

根据字数统计，可以很容易地识别和修剪稀有词。或者，他们的计数可以汇总到一个特殊的垃圾箱中，可以作为附加功能。图3-7展示了一个短文档中的表示形式，该短文档包含一些常用单词和两个稀有词 `"gobbledygook"` 和 `"zylophant"` 。通常单词保留自己的计数，可以通过停用词列表或其他频率进一步过滤方法。这些难得的单词会失去他们的身份并被分组到垃圾桶功能中.

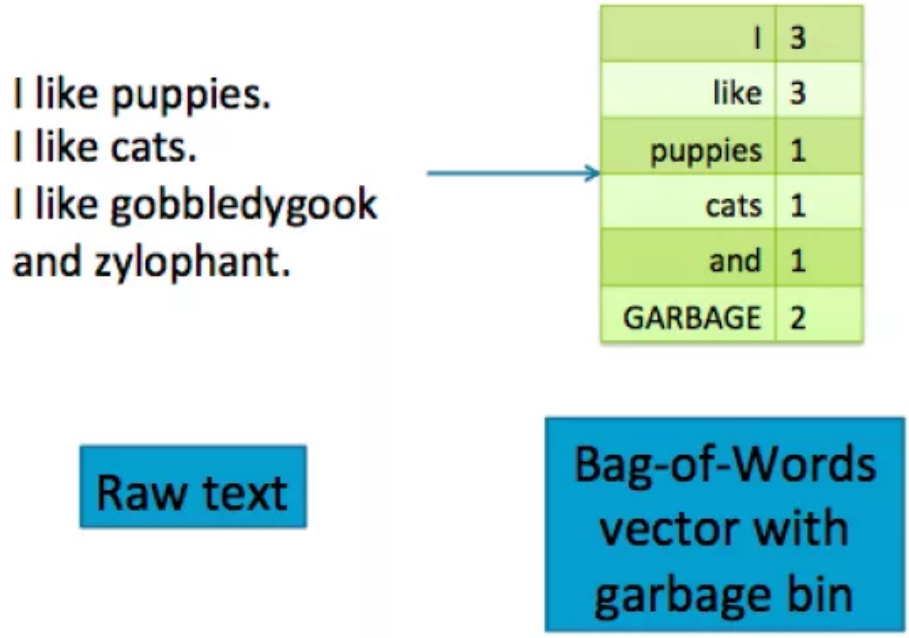


Figure 3-7. Bag-of-words feature vector with a garbage bin

由于在计算整个语料库之前不会知道哪些词很少，因此需要收集垃圾桶功能作为后处理步骤。

由于本书是关于特征工程的，因此我们将重点放在特征上。但稀有概念也适用于数据点。如果文本文档很短，那么它可能不包含有用的信息，并且在训练模型时不应使用该信息。

应用此规则时必须谨慎。维基百科转储包含许多不完整的存根，可能安全过滤。另一方面，推文本身就很短，并且需要其他特征和建模技巧。

词干解析（Stemming）

简单解析的一个问题是同一个单词的不同变体会被计算为单独的单词。例如，`"flower"`和`"flowers"`在技术上是不同的记号，`"swimmer"`，`"swimming"`和`"swim"`也是如此，尽管它们的含义非常接近。如果所有这些不同的变体都映射到同一个单词，那将会很好。

词干解析是一项 NLP 任务，试图将单词切分为基本的语言词干形式。有不同的方法。有些基于语言规则，其他基于观察统计。被称为词形化的算法的一个子类将词性标注和语言规则结合起来。

Porter stemmer 是英语中使用最广泛的免费词干工具。原来的程序是用 ANSI C 编写的，但是很多其他程序包已经封装它来提供对其他语言的访问。尽管其他语言的努力正在进行，但大多数词干工具专注于英语。

以下是通过 NLTK Python 包运行 Porter stemmer 的示例。正如我们所看到的，它处理了大量的情况，包括将`"sixties"`和`"sixty"`转变为同一根`"sixti"`。但这并不完美。单词`"goes"`映射到`"goe"`，而`"go"`映射到它自己。

```
import nltk
stemmer = nltk.stem.porter.PorterStemmer()
```

```
stemmer.stem('flowers')
```

```
'flower'
```

```
stemmer.stem('zeroes')
```

```
'zero'
```

```
stemmer.stem('stemmer')
```

```
'stemmer'
```

```
stemmer.stem('sixties')
```

```
'sixti'
```

```
stemmer.stem('sixty')
```

```
'sixti'
```

```
stemmer.stem('goes')
```

```
'goe'
```

```
stemmer.stem('go')
```

```
'go'
```

词干解析的确有一个计算成本。最终收益是否大于成本取决于应用程序。

含义的原子：从单词到 N-gram 到短语

词袋的概念很简单。但是，一台电脑怎么知道一个词是什么？文本文档以数字形式表示为一个字符串，基本上是一系列字符。也可能会遇到 JSON blob 或 HTML 页面形式的半结构化文本。但即使添加了标签和结构，基本单位仍然是一个字符串。如何将字符串转换为一系列的单词？这涉及解析和标记化的任务，我们将在下面讨论。

解析和分词

当字符串包含的不仅仅是纯文本时，解析是必要的。例如，如果原始数据是网页，电子邮件或某种类型的日志，则它包含额外的结构。人们需要决定如何处理日志中的标记，页眉，页脚或无趣的部分。如果文档是网页，则解析器需要处理 URL。如果是电子邮件，则可能需要特殊字段，例如 From，To 和 Subject 需要被特别处理，否则，这些标题将作为最终计数中的普通单词统计，这可能没有用处。

解析后，文档的纯文本部分可以通过标记。这将字符串（一系列字符）转换为一系列记号。然后可以将每个记号计为一个单词。分词器需要知道哪些字符表示一个记号已经结束，另一个正在开始。空格字符通常是好的分隔符，正如标点符号一样。如果文本包含推文，则不应将井号（`#`）用作分隔符（也称为分隔符）。

有时，分析需要使用句子而不是整个文档。例如，n-gram 是一个句子的概括，不应超出句子范围。更复杂的文本特征化方法，如 word2vec 也适用于句子或段落。在这些情况下，需要首先将文档解析为句子，然后将每个句子进一步标记为单词。

字符串对象

字符串对象有各种编码，如 ASCII 或 Unicode。纯英文文本可以用 ASCII 编码。一般语言需要 Unicode。如果文档包含非 ASCII 字符，则确保分词器可以处理该特定编码。否则，结果将不正确。

短语检测的搭配提取

连续的记号能立即被转化成词表和 n-gram。但从语义上讲，我们更习惯于理解短语，而不是 n-gram。在计算自然语言处理中，有用短语的概念被称为搭配。用 Manning 和 Schütze（1999：141）的话来说：“搭配是一个由两个或两个以上单词组成的表达，它们对应于某种常规的说话方式。”

搭配比其部分的总和更有意义。例如，`"strong tea"` 具有超越 `"great physical strength"` 和 `"tea"` 的不同含义，因此被认为是搭配。另一方面，“可爱的小狗”这个短语恰恰意味着它的部分总和：“可爱”和“小狗”。因此，它不被视为搭配。

搭配不一定是连续的序列。`"Emma knocked on the door"` 一词被认为包含搭配 `"knock door"`，因此不是每一个搭配都是一个 n-gram。相反，并不是每个 n-gram 都被认为是一个有意义的搭配。

由于搭配不仅仅是其部分的总和，它们的含义也不能通过单个单词计数来充分表达。作为一种表现形式，词袋不足。袋子的 ngram 也是有问题的，因为它们捕获了太多无意义的序列（考虑 `"this is in the bag-of-ngram example"`），而没有足够的有意义的序列。

搭配作为功能很有用。但是，如何从文本中发现并提取它们呢？一种方法是预先定义它们。如果我们努力尝试，我们可能会找到各种语言的全面成语列表，我们可以通过文本查看任何匹配。这将是非常昂贵的，但它会工作。如果语料库是非常特定领域的并且包含深奥的术语，那么这可能是首选的方法。但是这个列表需要大量的手动管理，并且需要不断更新语料库。例如，分析推文，博客和文章可能不太现实。

自从统计 NLP 过去二十年出现以来，人们越来越多地选择用于查找短语的统计方法。统计搭配提取方法不是建立固定的短语和惯用语言列表，而是依赖不断发展的数据来揭示当今流行的语言。

基于频率的方法

一个简单的黑魔法是频繁发生的 n-gram。这种方法的问题是最常发生的，这种可能不是最有用的。表 3-2 显示了整个 Yelp 评论数据集中最流行的 bigram（`n=2`）。正如我们所知的，按文

件计数排列的最常见的十大常见术语是非常通用的术语，并不包含太多含义。

Table 3-2. Most frequently occurring 2-grams in a Yelp reviews dataset

Bigram	Document Count
of the	450849
and the	426346
in the	397821
it was	396713
this place	344800
it s	341090
and i	332415
on the	325044
i was	285012
for the	276946

用于搭配提取的假设检验

原始流行度计数（Raw popularity count）是一个比较粗糙的方法。我们必须找到更聪慧的统计数据才能够轻松挑选出有意义的短语。关键的想法是看两个单词是否经常出现在一起。回答这个问题的统计机制被称为假设检验。

假设检验是将噪音数据归结为“是”或“否”的答案。它涉及将数据建模为从随机分布中抽取的样本。随机性意味着人们永远无法 100% 的确定答案；总会有异常的机会。所以答案附在概率上。例如，假设检验的结果可能是“这两个数据集来自同一分布，其概率为 95%”。对于假设检验的温和介绍，请参阅可汗学院关于假设检验和 p 值的教程。

在搭配提取的背景下，多年来已经提出了许多假设检验。最成功的方法之一是基于似然比检验（Dunning，1993）。对于给定的一对单词，该方法测试两个假设观察的数据集。假设 1（原假设）表示，词语 1 独立于词语 2 出现。另一种说法是说，看到词语1对我们是否看到词语2没有影响。假设 2（备选假设）说，看到词 1 改变了看到单词 2 的可能性。我们采用备选假设来暗示这两个单词形成一个共同的短语。因此，短语检测（也称为搭配提取）的似然比检验提出了以下问题：给定文本语料库中观察到的单词出现更可能是从两个单词彼此独立出现的模型中生成的，或者模型中两个词的概率纠缠？

这是有用的。让我们算一点。（数学非常精确和简洁地表达事物，但它确实需要与自然语言完全不同的分析器。）

Null hypothesis H_{null} (independent): $P(w_2 \mid w_1) = P(w_2 \mid \text{not } w_1)$

Alternate hypothesis $H_{alternate}$ (not independent): $P(w_2 \mid w_1) \neq P(w_2 \mid \text{not } w_1)$

The final statistic is the log of the ratio between the two:

$$\log \lambda = \log \frac{L(Data; H_{null})}{L(Data; H_{alternate})}.$$

似然函数 `L(Data; H)` 表示在单词对的独立模型或非独立模型下观察数据集中词频的概率。为了计算这个概率，我们必须对如何生成数据做出另一个假设。最简单的数据生成模型是二项模型，其中对于数据集中的每个单词，我们抛出一个硬币，并且如果硬币朝上出现，我们插入我们的特殊单词，否则插入其他单词。在此策略下，特殊词的出现次数遵循二项分布。二项分布完全由词的总数，词的出现次数和词首概率决定。

似然比检验分析常用短语的算法收益如下。

1. 计算所有单体词的出现概率： `p(w)` 。
2. 计算所有唯一双元的条件成对词发生概率： `p(W2 × W1)`
3. 计算所有唯一的双对数似然比对数。
4. 根据它们的似然比排序双字节。
5. 以最小似然比值作为特征。

掌握似然比测试

关键在于测试比较的不是概率参数本身，而是在这些参数（以及假设的数据生成模型）下观察数据的概率。可能性是统计学习的关键原则之一。但是在你看到它的前几次，这绝对是一个令人困惑的问题。一旦你确定了逻辑，它就变得直观了。

还有另一种基于点互信息的统计方法。但它对真实世界文本语料库中常见的罕见词很敏感。因此它不常用，我们不会在这里展示它。

请注意，搭配抽取的所有统计方法，无论是使用原始频率，假设测试还是点对点互信息，都是通过过滤候选词组列表来进行操作的。生成这种清单的最简单和最便宜的方法是计算 n-gram。它可能产生不连续的序列，但是它们计算成本颇高。在实践中，即使是连续 n-gram，人们也很少超过 bi-gram 或 tri-gram，因为即使在过滤之后，它们的数量也很多。为了生成更长的短语，还有其他方法，如分块或与词性标注相结合。

分块（Chunking）和词性标注（part-of-Speech Tagging）

分块比 n-gram 要复杂一点，因为它基于词性，基于规则的模型形成了记号序列。

例如，我们可能最感兴趣的是在问题中找到所有名词短语，其中文本的实体，主题最为有趣。为了找到这个，我们使用词性标记每个作品，然后检查该标记的邻域以查找词性分组或“块”。定义单词到词类的模型通常是语言特定的。几种开源 Python 库（如 NLTK，Spacy 和 TextBlob）具有多种语言模型。

为了说明 Python 中的几个库如何使用词性标注非常简单地进行分块，我们再次使用 Yelp 评论数据集。我们将使用 spacy 和 TextBlob 来评估词类以找到名词短语。

```
import pandas as pd
import json
# Load the first 10 reviews
f = open('data/yelp_academic_dataset_review.json')
js = []
for i in range(10):
    js.append(json.loads(f.readline()))
f.close()
review_df = pd.DataFrame(js)
## First we'll walk through spaCy's functions
```

```
# !pip install -U spacy
# !python -m spacy download en#使用spacy，需要安装上述两个步骤，去掉注释即可运行
```

```
import spacy
# preload the language model
nlp = spacy.load('en')
# We can create a Pandas Series of spaCy nlp variables
doc_df = review_df['text'].apply(nlp)
# spaCy gives you fine grained parts of speech using: (.pos_)
# and coarse grained parts of speech using: (.tag_)
for doc in doc_df[4]:
    print([doc.text, doc.pos_, doc.tag_])
```

```
['General', 'PROPN', 'NNP']
['Manager', 'PROPN', 'NNP']
['Scott', 'PROPN', 'NNP']
['Petello', 'PROPN', 'NNP']
['is', 'VERB', 'VBZ']
['a', 'DET', 'DT']
['good', 'ADJ', 'JJ']
['egg', 'NOUN', 'NN']
```

```
['!', 'PUNCT', '.']
['!', 'PUNCT', '.']
['!', 'PUNCT', '.']
['Not', 'ADV', 'RB']
['to', 'PART', 'TO']
['go', 'VERB', 'VB']
['into', 'ADP', 'IN']
['detail', 'NOUN', 'NN']
[',', 'PUNCT', ',']
['but', 'CCONJ', 'CC']
['let', 'VERB', 'VB']
['me', 'PRON', 'PRP']
['assure', 'VERB', 'VB']
['you', 'PRON', 'PRP']
['if', 'ADP', 'IN']
['you', 'PRON', 'PRP']
['have', 'VERB', 'VBP']
['any', 'DET', 'DT']
['issues', 'NOUN', 'NNS']
['(', 'PUNCT', '-LRB-']
['albeit', 'ADP', 'IN']
['rare', 'ADJ', 'JJ']
[')', 'PUNCT', '-RRB-']
['speak', 'VERB', 'VBP']
['with', 'ADP', 'IN']
['Scott', 'PROPN', 'NNP']
['and', 'CCONJ', 'CC']
['treat', 'VERB', 'VB']
['the', 'DET', 'DT']
['guy', 'NOUN', 'NN']
['with', 'ADP', 'IN']
['some', 'DET', 'DT']
['respect', 'NOUN', 'NN']
['as', 'ADP', 'IN']
['you', 'PRON', 'PRP']
['state', 'VERB', 'VBP']
['your', 'DET', 'PRP$']
['case', 'NOUN', 'NN']
['and', 'CCONJ', 'CC']
['I', 'PRON', 'PRP']
["'d", 'AUX', 'MD']
['be', 'VERB', 'VB']
['surprised', 'ADJ', 'JJ']
['if', 'ADP', 'IN']
['you', 'PRON', 'PRP']
['do', 'VERB', 'VBP']
["n't", 'ADV', 'RB']
['walk', 'VERB', 'VB']
['out', 'ADV', 'RB']
['totally', 'ADV', 'RB']
['satisfied', 'ADJ', 'JJ']
['as', 'ADP', 'IN']
['I', 'PRON', 'PRP']
['just', 'ADV', 'RB']
['did', 'VERB', 'VBD']
```

```
['.', 'PUNCT', '.']
['Like', 'INTJ', 'UH']
['I', 'PRON', 'PRP']
['always', 'ADV', 'RB']
['say', 'VERB', 'VBP']
['.....', 'PUNCT', '.']
['"', 'PUNCT', "''"]
['Mistakes', 'NOUN', 'NNS']
['are', 'VERB', 'VBP']
['inevitable', 'ADJ', 'JJ']
[',', 'PUNCT', ',']
['it', 'PRON', 'PRP']
["'s", 'VERB', 'VBZ']
['how', 'ADV', 'WRB']
['we', 'PRON', 'PRP']
['recover', 'VERB', 'VBP']
['from', 'ADP', 'IN']
['them', 'PRON', 'PRP']
['that', 'DET', 'WDT']
['is', 'VERB', 'VBZ']
['important', 'ADJ', 'JJ']
['"', 'PUNCT', "''"]
['!', 'PUNCT', '.']
['!', 'PUNCT', '.']
['!', 'PUNCT', '.']
['\n\n', 'SPACE', '_SP']
['Thanks', 'NOUN', 'NNS']
['to', 'ADP', 'IN']
['Scott', 'PROPN', 'NNP']
['and', 'CCONJ', 'CC']
['his', 'DET', 'PRP$']
['awesome', 'ADJ', 'JJ']
['staff', 'NOUN', 'NN']
['.', 'PUNCT', '.']
['You', 'PRON', 'PRP']
["'ve", 'VERB', 'VB']
['got', 'VERB', 'VBN']
['a', 'DET', 'DT']
['customer', 'NOUN', 'NN']
['for', 'ADP', 'IN']
['life', 'NOUN', 'NN']
['!', 'PUNCT', '.']
['!', 'PUNCT', '.']
['..........', 'PUNCT', '.']
[':', 'PUNCT', ':']
['^', 'PUNCT', 'LS']
[')', 'PUNCT', '-RRB-']
```

```
print([chunk for chunk in doc_df[4].noun_chunks])
```

```
[General Manager Scott Petello, a good egg, detail, me, you, you, any issues, Scott, the guy, some resp
```

```
import nltk
```

```
nltk.download('averaged_perceptron_tagger')
```

True

```
## We can do the same feature transformations using Textblob
from textblob import TextBlob
# The default tagger in TextBlob uses the PatternTagger, which is fine for our example.
# You can also specify the NLTK tagger, which works better for incomplete sentences.
blob_df = review_df['text'].apply(TextBlob)
blob_df[4].tags
```

```
[('General', 'NNP'),
 ('Manager', 'NNP'),
 ('Scott', 'NNP'),
 ('Petello', 'NNP'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('good', 'JJ'),
 ('egg', 'NN'),
 ('Not', 'RB'),
 ('to', 'TO'),
 ('go', 'VB'),
 ('into', 'IN'),
 ('detail', 'NN'),
 ('but', 'CC'),
 ('let', 'VB'),
 ('me', 'PRP'),
 ('assure', 'VB'),
 ('you', 'PRP'),
 ('if', 'IN'),
 ('you', 'PRP'),
 ('have', 'VBP'),
 ('any', 'DT'),
 ('issues', 'NNS'),
 ('albeit', 'IN'),
 ('rare', 'NN'),
 ('speak', 'NN'),
 ('with', 'IN'),
 ('Scott', 'NNP'),
 ('and', 'CC'),
 ('treat', 'VB'),
 ('the', 'DT'),
 ('guy', 'NN'),
 ('with', 'IN'),
 ('some', 'DT'),
 ('respect', 'NN'),
 ('as', 'IN'),
 ('you', 'PRP'),
 ('state', 'NN'),
 ('your', 'PRP$'),
 ('case', 'NN'),
```

```
('and', 'CC'),
('I', 'PRP'),
("'d", 'MD'),
('be', 'VB'),
('surprised', 'VBN'),
('if', 'IN'),
('you', 'PRP'),
('do', 'VBP'),
("n't", 'RB'),
('walk', 'VB'),
('out', 'RP'),
('totally', 'RB'),
('satisfied', 'JJ'),
('as', 'IN'),
('I', 'PRP'),
('just', 'RB'),
('did', 'VBD'),
('Like', 'IN'),
('I', 'PRP'),
('always', 'RB'),
('say', 'VBP'),
('..', 'VBP'),
('Mistakes', 'NNS'),
('are', 'VBP'),
('inevitable', 'JJ'),
('it', 'PRP'),
("'s", 'VBZ'),
('how', 'WRB'),
('we', 'PRP'),
('recover', 'VBP'),
('from', 'IN'),
('them', 'PRP'),
('that', 'WDT'),
('is', 'VBZ'),
('important', 'JJ'),
('Thanks', 'NNS'),
('to', 'TO'),
('Scott', 'NNP'),
('and', 'CC'),
('his', 'PRP$'),
('awesome', 'JJ'),
('staff', 'NN'),
('You', 'PRP'),
("'ve", 'VBP'),
('got', 'VBN'),
('a', 'DT'),
('customer', 'NN'),
('for', 'IN'),
('life', 'NN'),
('^', 'NN')]
```

```
print([np for np in blob_df[4].noun_phrases])
```

```
['general manager', 'scott petello', 'good egg', 'scott', "n't walk", '... ..', 'mistakes', 'thanks',
```

你可以看到每个库找到的名词短语有些不同。spacy 包含英语中的常见单词，如 `"a"` 和 `"the"`，而 TextBlob 则删除这些单词。这反映了规则引擎的差异，它驱使每个库都认为是“名词短语”。你也可以写你的词性关系来定义你正在寻找的块。使用 Python 进行自然语言处理可以深入了解从头开始用 Python 进行分块。

案例

构造一个文本数据集

```
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'animals', 'weather', 'animals']
corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus,
                          'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]
corpus_df
```

	Document	Category
0	The sky is blue and beautiful.	weather
1	Love this blue and beautiful sky!	weather
2	The quick brown fox jumps over the lazy dog.	animals
3	The brown fox is quick and the blue dog is lazy!	animals
4	The sky is very blue and the sky is very beaut...	weather
5	The dog is lazy but the brown fox is quick!	animals

基本预处理

```
nltk.download()#
```

True

```
#词频与停用词
wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')
print (stop_words)
def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z0-9\s]', '', doc, re.I)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)
```

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "y

```
norm_corpus = normalize_corpus(corpus)
norm_corpus
#The sky is blue and beautiful.
```

```
array(['sky blue beautiful', 'love blue beautiful sky',
       'quick brown fox jumps lazy dog', 'brown fox quick blue dog lazy',
       'sky blue sky beautiful today', 'dog lazy brown fox quick'],
      dtype='<U30')
```

词袋模型

```
from sklearn.feature_extraction.text import CountVectorizer
print (norm_corpus)
cv = CountVectorizer(min_df=0., max_df=1.)
cv.fit(norm_corpus)
print (cv.get_feature_names())
cv_matrix = cv.fit_transform(norm_corpus)
```

```
cv_matrix = cv_matrix.toarray()
cv_matrix
```

```
['sky blue beautiful' 'love blue beautiful sky'
 'quick brown fox jumps lazy dog' 'brown fox quick blue dog lazy'
 'sky blue sky beautiful today' 'dog lazy brown fox quick']
['beautiful', 'blue', 'brown', 'dog', 'fox', 'jumps', 'lazy', 'love', 'quick', 'sky', 'today']

array([[1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0],
       [0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 0, 2, 1],
       [0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0]], dtype=int64)
```

```
vocab = cv.get_feature_names()
pd.DataFrame(cv_matrix, columns=vocab)
```

	beautiful	blue	brown	dog	fox	jumps	lazy	love	quick	sky	today
0	1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	1	0	1	0
2	0	0	1	1	1	1	1	0	1	0	0
3	0	1	1	1	1	0	1	0	1	0	0
4	1	1	0	0	0	0	0	0	0	2	1
5	0	0	1	1	1	0	1	0	1	0	0

N-Grams模型

```
bv = CountVectorizer(ngram_range=(2,2))
bv_matrix = bv.fit_transform(norm_corpus)
bv_matrix = bv_matrix.toarray()
vocab = bv.get_feature_names()
pd.DataFrame(bv_matrix, columns=vocab)
```

	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	brown fox	dog lazy	fox jumps	fox quick	jumps lazy	lazy brown	lazy dog	love blue	quick blue	quick brown	sky beautiful	sky blue
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	0
3	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	0	0
4	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
5	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0

TF-IDF 模型

```
from sklearn.feature_extraction.text import TfidfVectorizer #中国 蜜蜂 养殖 它们的片频数都是20次
tv = TfidfVectorizer(min_df=0., max_df=1., use_idf=True)
tv_matrix = tv.fit_transform(norm_corpus)
tv_matrix = tv_matrix.toarray()

vocab = tv.get_feature_names()
pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```

	beautiful	blue	brown	dog	fox	jumps	lazy	love	quick	sky	today
0	0.60	0.52	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00
1	0.46	0.39	0.00	0.00	0.00	0.00	0.00	0.66	0.00	0.46	0.00
2	0.00	0.00	0.38	0.38	0.38	0.54	0.38	0.00	0.38	0.00	0.00
3	0.00	0.36	0.42	0.42	0.42	0.00	0.42	0.00	0.42	0.00	0.00

	beautiful	blue	brown	dog	fox	jumps	lazy	love	quick	sky	today
4	0.36	0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.52
5	0.00	0.00	0.45	0.45	0.45	0.00	0.45	0.00	0.45	0.00	0.00

Similarity特征

```
from sklearn.metrics.pairwise import cosine_similarity

similarity_matrix = cosine_similarity(tv_matrix)
similarity_df = pd.DataFrame(similarity_matrix)
similarity_df
```

	0	1	2	3	4	5
0	1.000000	0.753128	0.000000	0.185447	0.807539	0.000000
1	0.753128	1.000000	0.000000	0.139665	0.608181	0.000000
2	0.000000	0.000000	1.000000	0.784362	0.000000	0.839987
3	0.185447	0.139665	0.784362	1.000000	0.109653	0.933779
4	0.807539	0.608181	0.000000	0.109653	1.000000	0.000000
5	0.000000	0.000000	0.839987	0.933779	0.000000	1.000000

聚类特征

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=2)
km.fit_transform(similarity_df)
cluster_labels = km.labels_
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```

	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	1

	Document	Category	ClusterLabel
1	Love this blue and beautiful sky!	weather	1
2	The quick brown fox jumps over the lazy dog.	animals	0
3	The brown fox is quick and the blue dog is lazy!	animals	0
4	The sky is very blue and the sky is very beaut...	weather	1
5	The dog is lazy but the brown fox is quick!	animals	0

主题模型

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_topics=2, max_iter=100, random_state=42)
dt_matrix = lda.fit_transform(tv_matrix)
features = pd.DataFrame(dt_matrix, columns=['T1', 'T2'])
features
```

	T1	T2
0	0.190548	0.809452
1	0.176804	0.823196
2	0.846184	0.153816
3	0.814863	0.185137
4	0.180516	0.819484
5	0.839172	0.160828

主题和词的权重

```
tt_matrix = lda.components_
for topic_weights in tt_matrix:
    topic = [(token, weight) for token, weight in zip(vocab, topic_weights)]
    topic = sorted(topic, key=lambda x: -x[1])
    topic = [item for item in topic if item[1] > 0.6]
    print(topic)
    print()
```

```
[('brown', 1.7273638692668465), ('dog', 1.7273638692668465), ('fox', 1.7273638692668465), ('lazy', 1.

[('sky', 2.264386643135622), ('beautiful', 1.9068269319456903), ('blue', 1.7996282104933266), ('love'
```

词嵌入模型

```
from gensim.models import word2vec

wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_corpus]

# Set values for various parameters
feature_size = 10    # Word vector dimensionality
window_context = 10          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

w2v_model = word2vec.Word2Vec(tokenized_corpus, size=feature_size,
                          window=window_context, min_count = min_word_count,
                          sample=sample)
```

```
w2v_model.wv['sky']
```

```
array([ 0.04776765, -0.04441591,  0.0468228 , -0.04031719, -0.04735648,
       -0.00321561, -0.03345697, -0.0451241 ,  0.03330296, -0.03037446],
      dtype=float32)
```

```
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector
```

```
def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary, num_features)
                for tokenized_sentence in corpus]
    return np.array(features)
```

```
w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus, model=w2v_model,
                                             num_features=feature_size)
pd.DataFrame(w2v_feature_array) #lstm
```

	0	1	2	3	4	5	6	7	8	9
0	0.024127	0.017077	0.026422	-0.029402	0.001112	-0.002655	-0.004409	-0.008712	0.009802	0.012457
1	0.029736	0.022432	0.018225	-0.022848	-0.010878	-0.004652	-0.012399	-0.000851	-0.003157	0.018775
2	-0.002641	-0.005180	0.005764	-0.001435	0.018988	0.010134	0.001064	-0.004431	-0.002646	0.000689
3	-0.000735	-0.004496	0.008730	-0.011999	0.018633	0.017814	-0.000056	-0.001533	0.001793	0.016195
4	0.030725	-0.006659	0.023489	-0.032785	-0.000014	-0.010089	-0.016119	-0.005245	0.009107	0.007049
5	-0.001140	-0.015327	0.001268	-0.008928	0.012809	0.013047	-0.001205	-0.001266	-0.003366	0.010224

总结

词袋模型易于理解和计算，对分类和搜索任务很有用。但有时单个单词太简单，不足以将文本中的某些信息封装起来。为了解决这个问题，人们寄希望于比较长的序列。Bag-of-ngram 是 BOW 的自然概括，这个概念仍然容于理解，而且它的计算开销这就像 BOW 一样容易。

Bag of-ngram 生成更多不同的 ngram。它增加了特征存储成本，以及模型训练和预测阶段的计算成本。虽然数据点的数量保持不变，但特征空间的维度现在更大。因此数据密度更为稀疏。n 越高，存储和计算成本越高，数据越稀疏。由于这些原因，较长的 n-gram 并不总是会使模型精度的得到提高（或任何其他性能指标）。人们通常在 `n = 2` 或 3 时停止。较少的 n-gram 很少被使用。

防止稀疏性和成本增加的一种方法是过滤 n-gram 并保留最有意义的短语。这是搭配抽取的目标。理论上，搭配（或短语）可以在文本中形成非连续的标记序列。然而，在实践中，寻找非连续词组的计算成本要高得多并且没有太多的收益。因此搭配抽取通常从一个候选人名单中开始，并利用统计方法对他们进行过滤。

所有这些方法都将一系列文本标记转换为一组断开的计数。与一个序列相比，一个集合的结构要少得多；他们导致平面特征向量。

在本章中，我们用简单的语言描述文本特征化技术。这些技术将一段充满丰富语义结构的自然语言文本转化为一个简单的平面向量。我们讨论一些常用的过滤技术来降低向量维度。我们还引入了 ngram 和搭配抽取作为方法，在平面向量中添加更多的结构。下一章将详细介绍另一种常见的文本特征化技巧，称为 tf-idf。随后的章节将讨论更多方法将结构添加回平面向量。

参考文献

Dunning, Ted. 1993. “Accurate methods for the statistics of surprise and

coincidence.” ACM Journal of Computational Linguistics, special issue on using large corpora , 19:1 (61—74).

“Hypothesis Testing and p-Values.” Khan Academy, accessed May 31,

2016,https://www.khanacademy.org/math/probability/statistics-inferential/hypothesis-testing/v/hypothesis-testing-and-p-values.

Manning,Christopher D. and Hinrich Schütze. 1999. Foundations of StatisticalNatural Language Processing . Cambridge, Massachusettes: MIT Press.

Sometimes people call it the document “vector.” The vector extends from the original and ends at the specified point. For our purposes, “vector” and “point” are the same thing.

相关资源

原版（英文）图书地址:

https://www.oreilly.com/library/view/feature-engineering-for/9781491953235/

本文代码可以在github下载：

https://github.com/fengdu78/Data-Science-Notes/tree/master/9.feature-engineering

数据集的百度云：

链接：https://pan.baidu.com/s/1uDXt5jWUOfI0fS7hD91vBQ 提取码：8p5d

往期精彩回顾

- 那些年做的学术公益-你不是一个人在战斗
- 适合初学者入门人工智能的路线及资料下载
- 机器学习在线手册
- 深度学习在线手册

备注：加入本站微信群或者qq群，请回复“加群”

加入知识星球（4500+用户，ID：92416895），请回复“知识星球”