**NLP\_Chapter5**

**第 5 章 分类和标注词汇**

标注是典型的 NLP 流水线中继分词之后的第二个步骤。

将词汇按它们的**词性**(parts-of-speech，POS)分类以及相应的标注它们的过程被称为**词性标注**(part-of-speech tagging, POS tagging)或干脆简称**标注**。词性也称为**词类**或**词汇范畴**。用于特定任务的标记的集合被称为一个**标记集**。

**5.1 使用词性标注器**

一个**词性标注器**(part-of-speech tagger 或 POS tagger)处理一个词序列，为每个词附加一个词性标记:

text = nltk.word\_tokenize("And now for something completely different")

print(nltk.pos\_tag(text))

[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]

在这里我们看到 and 是 CC，并列连词;now 和 completely 是 RB，副词;for 是 IN，介 词;something 是 NN，名词;different 是 JJ，形容词。

NLTK中提供了每个标记的文档，可以使用标记来查询，如:nltk.help.u penn\_tagset('RB')，或正则表达式，如:nl tk.help.upenn\_brown\_tagset('N N.\*')。一些语料库有标记集文档的 README 文件;见 nltk.name.readme ()，用语料库的名称替换 name。

包括一些同形同音异义词的例子:

text = nltk.word\_tokenize("They refuse to permit us to obtain the refuse permit")

print(nltk.pos\_tag(text))

[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]

句中的refuse和permit同形异义因而发音不同(出于这个原因，文本转语音系统通常要进行词性标注。)

用text.similar()方法为一个词 w 找出所有上下文 w1ww2，然后找出所有出现在相同上下文中的词 w'，即w1w'w2:

text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())

text.similar('woman')

text.similar('bought')

text.similar('over')

text.similar('the')

一个标注器能够正确识别一个句子的上下文中的这些词的标记。

一个标注器也可以为我们对未知词的认识过程建模;例如:我们可以根据词根 scrobble 猜测 scrobbling 可能是一个动词，并有可能发生在 he was scrobbling 这样的上下文中。

**5.2 标注语料库**

**表示已标注的标识符**

按照 NLTK 的约定，一个已标注的标识符使用一个由标识符和标记组成的元组来表示。可以使用函数 str2tuple()从表示一个已标注的标识符的标准字符串创建一个这样的特殊元组:

tagged\_token = nltk.tag.str2tuple('fly/NN')

print(tagged\_token)

print(tagged\_token[0])

print(tagged\_token[1])

可以直接从一个字符串构造一个已标注的标识符的链表。第一步是对字符串分词以便能访问单独的词/标记字符串，然后将每一个转换成一个元组(使用 str2tuple()):

print([nltk.tag.str2tuple(t) for t in sent.split()])

**读取已标注的语料库**

NLTK 中包括的若干语料库**已标注**了词性。

不同语料库使用不同格式存储词性标记。NLTK 中的语料库阅读器提供了一个统一的接口，使你不必理会这些不同的文件格式。

布朗语料库的语料库阅读器按如下所示的方式表示数据：

print(nltk.corpus.brown.tagged\_words()[:50])

print(nltk.corpus.brown.tagged\_words(tagset='universal')[:50])

只要语料库包含已标注的文本，NLTK的语料库接口都将有一个 tagged\_words()方法:

print(nltk.corpus.nps\_chat.tagged\_words()[:50])

print(nltk.corpus.conll2000.tagged\_words()[:50])

print(nltk.corpus.treebank.tagged\_words()[:50])

并非所有的语料库都采用同一组标记;看前面提到的标记集的帮助函数和 readme()方法中的文档。想避免这些标记集的复杂化，使用一个内置的到一个简化的标记集的映射:

print(nltk.corpus.brown.tagged\_words(tagset='universal')[:50])

print(nltk.corpus.treebank.tagged\_words(tagset='universal')[:50])

NLTK 中还有其他几种语言的已标注语料库，包括中文，印地语，葡萄牙语，西班牙语，荷兰语和加泰罗尼亚语。这些通常含有非 ASCII 文本，当输出较大的结构如列表时，Python 总是以十六进制显示这些：

print(nltk.corpus.sinica\_treebank.tagged\_words()[:20])

print(nltk.corpus.indian.tagged\_words()[:20])

print(nltk.corpus.mac\_morpho.tagged\_words()[:20])

print(nltk.corpus.conll2002.tagged\_words()[:20])

print(nltk.corpus.cess\_cat.tagged\_words()[:20])

如果环境设置正确，有适合的编辑器和字体，应该能够以人可读的方式显示单个字符串。

如果语料库也被分割成句子，将有一个 tagged\_sents()方法将已标注的词划分成句子，而不是将它们表示成一个大链表。

**简化的词性标记集**

简化的标记集

标记 含义 例子

ADJ adjective new, good, high, special, big, local

ADP adposition on, of, at, with, by, into, under

ADV adverb really, already, still, early, now

CONJ conjunction and, or, but, if, while, although

DET determiner, article the, a, some, most, every, no, which

NOUN noun year, home, costs, time, Africa

NUM numeral twenty-four, fourth, 1991, 14:24

PRT particle at, on, out, over per, that, up, with

PRON pronoun he, their, her, its, my, I, us

VERB verb is, say, told, given, playing, would

. punctuation marks . , ; !

X other ersatz, esprit, dunno, gr8, university

看看这些标记中哪些是布朗语料库的新闻类中最常见的:

brown\_news\_tagged = brown.tagged\_words(categories='news',tagset='universal')

tag\_fd = nltk.FreqDist(tag for (word,tag) in brown\_news\_tagged)

print(tag\_fd.most\_common(10))

tag\_fd.plot(cumulative=True)

可以使用这些标记做强大的搜索，结合一个图形化的 POS 一致性工具 nltk.app.concordance()。用它来寻找任一词和 POS 标记的组合，如:N N N N, hit/VD, hit/VN 或the ADJ man。

**名词**

名词一般指的是人、地点、事情或概念，名词可能出现在限定词和形容词之后，可以是动词的主语或宾语

简化的名词标记对普通名词是 N，如:书，对专有名词是 NP，如苏格兰。

检查一些已标注的文本，看看哪些词类出现在一个名词前，频率最高的在最前面:

word\_tag\_pairs = nltk.bigrams(brown\_news\_tagged)

fdist=nltk.FreqDist([a[1] for (a,b) in word\_tag\_pairs if b[1] =='NOUN'])

print([tag for (tag,\_) in fdist.most\_common()])

首先，构建一个双连词链表，它的成员是它们自己的词-标记对。然后，构建一个双连词的标记部分的 FreqDist。

这证实了我们的断言:名词出现在限定词和形容词之后，包括数字形容词(数词，标注为N U M )。

**动词**

动词是用来描述事件和行动的词, 在一个句子中，动词通常表示涉及一个或多个名词短语所指示物的关系。

找出新闻文本中最常见的动词:

wsj = nltk.corpus.treebank.tagged\_words(tagset='universal')

word\_tag\_fd = nltk.FreqDist(wsj)

print([wt[0] for (wt,\_) in word\_tag\_fd.most\_common(50) if wt[1] == 'VERB'])

频率分布中计算的项目是词-标记对。由于词汇和标记是成对的，我们可以把词作为条件，标记作为事件，使用条件-事件对的链表初始化一个条件频率分布。这让我们看到了一个给定的词的标记的频率顺序列表:

cfd1 = nltk.ConditionalFreqDist(wsj)

print(cfd1['yield'].most\_common())

print(cfd1['cut'].most\_common())

我们可以颠倒配对的顺序，这样标记作为条件，词汇作为事件。现在我们可以看到对于一个给定的标记可能的词:

wsj = nltk.corpus.treebank.tagged\_words()

cfd2 = nltk.ConditionalFreqDist((tag,word) for (word,tag) in wsj)

print(cfd2['VBN'].most\_common(20))

要弄清 VD(过去式)和 VN(过去分词)之间的区别，让我们找到可以同是 VD 和 VN 的词汇，看看一些它们周围的文字:

cfd1 = nltk.ConditionalFreqDist(wsj)

print([w for w in cfd1.conditions() if 'VBD' in cfd1[w] and 'VBN' in cfd1[w]][:20])

idx1 = wsj.index(('kicked','VBD'))

print(wsj[idx1-4:idx1+1])

idx2 = wsj.index(('kicked','VBN'))

print(wsj[idx2-4:idx2+1])

指定一个过去分词的链表，收集所有直接在链表中项目前面的词-标记对:

vn = list(cfd2['VBN'])

for w in vn[:20]:

idx3 = wsj.index((w,'VBN'))

print(wsj[idx3-1])

**形容词和副词**

形容词修饰名词，可以作为修饰符或谓语，英语形容词可以有内部结构

副词修饰动词，指定时间、方式、地点或动词描述的事件的方向，副词也可以修饰形容词

英语中还有几个封闭的词类，如介词，冠词(也常称为限定词)(如:the，a)，情态动词(如:should，may)，人称代词 (如:she，they)。每个词典和语法对这些词的分类都不同。

**未简化的标记**

找出每个名词类型中最频繁的名词。下例中的程序找出所有以 NN 开始的标记，并为每个标记提供了几个示例词汇：

def findtags(tag\_prefix,tagged\_text):

cfd = nltk.ConditionalFreqDist((tag,word) for (word,tag) in tagged\_text

if tag.startswith(tag\_prefix))

return dict((tag,cfd[tag].most\_common(5)) for tag in cfd.conditions())

tagdict = findtags('NN',brown.tagged\_words(categories='news'))

for tag in sorted(tagdict):

print(tag,tagdict[tag])

可以看到有许多名词的变种;最重要的含有$的名词所有格,含有 S 的复数名词(因为复数名词通常以 s 结尾)，以及含有 P 的专有名词。此外，大多数的标记都有后缀修饰符:-NC 表示引用，-HL 表示headline中的词，-TL表示title(布朗标记的特征)。

**探索已标注的语料库**

假设我们正在研究词 often，想看看它是如何在文本中使用的。我们可以试着看看跟在 often 后面的词汇:

brown\_learned\_text = brown.words(categories='learned')

print(sorted(set(b for (a,b) in nltk.bigrams(brown\_learned\_text) if a=='often'))[:20])

然而，它使用 tagged\_words()方法查看跟随词的词性标记可能更有指导性:

brown\_lrnd\_tagged = brown.tagged\_words(categories='learned',tagset='universal')

tags = [b[1] for (a,b) in nltk.bigrams(brown\_lrnd\_tagged) if a[0] == 'often']

fd = nltk.FreqDist(tags)

fd.tabulate()

可以看到 often 后面最高频率的词性是动词。名词从来没有在这个位置出现(在这个特别

的语料中)。

在一些较大范围的上下文中，找出涉及特定标记和词序列的词。考虑句子中的每个三词窗口，检查它们是否符合标准。如果标记匹配，输出对应的词：

def process(sentence):

for (w1,t1),(w2,t2),(w3,t3) in nltk.trigrams(sentence):

if (t1.startswith('V') and t2 == 'TO' and t3.startswith('V')):

print(w1,w2,w3)

for tagged\_sent in brown.tagged\_sents()[:50]:

process(tagged\_sent)

找出与它们的标记关系高度模糊不清的词(有许多个不同标记的词)。了解为什么要标注这样的词是因为它们各自的上下文可以帮助我们弄清楚标记之间的区别：

brown\_news\_tagged = brown.tagged\_words(categories='news',tagset='universal')

data = nltk.ConditionalFreqDist((word.lower(),tag)

for (word,tag) in brown\_news\_tagged)

for word in data.conditions():

if len(data[word])>3:

tags = [tag for (tag,\_) in data[word].most\_common()]

print(word,' '.join(tags))

打开 POS 一致性工具 nltk.app.concordance()和加载完整的布朗语 料库(简化标记集)。

挑选一些上面代码例子末尾处列出的词，看看词的标记如何与词的上下文相关。例如:搜索 near 会看到所有混合在一起的形式， 搜索 near/ADJ 会看到它作为形容词使用，near N 会看到只是名词跟在后面的情况，等等。

**5.3 使用 Python 字典映射词及其属性**

一旦开始做词性标注，将会创建分配一个标记给一个词的程序，标记是在给定上下文中最可能的标记。可以认为这个过程是从词到标记的**映射**。在 Python 中最自然的方式存储映射是使用所谓的**字典**数据类型(在其他的编程语言又称为**关联数组**或**哈希数组** )。

**索引链表 VS 字典**

语言学对象从**键**到**值**的映射

语言学对象 映射来自 映射到

文档索引 词 页面列表(找到词的地方)

同义词 词意 同义词列表

词典 中心词 词条项(词性、意思定义、词源)

比较单词列表 注释术语 同源词(词列表，每种语言一个)

词形分析 表面形式 形态学分析(词素组件列表)

大多数情况下，从一个“词”映射到一些结构化对象。

**Python 字典**

Python 提供了一个**字典**数据类型，可用来做任意类型之间的映射。

定义 pos为一个空字典，然后给它添加四个项目，指定一些词的词性。使用熟悉的方括号将条目添加到字典：

pos = {}

print(pos)

pos['colorless'] = 'ADJ'

print(pos)

pos['ideas'] = 'N'

pos['sleep'] = 'V'

pos['furiously'] = 'ADV'

print(pos)

检查 pos 的值：键-值对的集合。一旦以这样的方式填充了字典，就可以使用键来检索值:

print(pos['ideas'])

print(pos['colorless'])

如果使用了一个尚未分配值的键，会报KeyError

字典中键值对的顺序与最初放入它们的顺序不同，这是因为字典不是序列而是映射，它的键 并不按固有的顺序。

如何算出一个字典的合法键?

如果字典不是太大，可以简单地通过查看变量 pos 检查它的内容。

另外，要找到键，我们可以将字典转换成一个链表或在需要使用链表的地方使用字典：

print(list(pos))

print(sorted(pos))

print([w for w in pos if w.endswith('s')])

与使用一个 for 循环遍历字典中的所有键一样，可以使用 for 循环输出字典的内容:

for word in sorted(pos):

print(word+':',pos[word])

这里“word+‘：’”可以使第一个词和：之间没有空格

字典的方法 keys()、values()和 items()允许我们访问作为单独的链表的键、值以及键-值对。甚至可以按它们的第一个元素排序元组�(如果第一个元素相同，就 使用它们的第二个元素)：

print(list(pos.keys()))

print(list(pos.values()))

print(list(pos.items()))

for key,val in sorted(pos.items()):

print(key+':',val)

字典中一个键只能对应一个值，但是有一个方法可以在该项目中存储多个值:使用一个链表值，例如:pos['sleep'] = ['N', 'V']。

**定义字典**

两种方式使用键-值对格式创建字典,通常使用第一个:

pos = {'colerless':'ADJ','ideas':'N','sleep':'V','furiously': 'ADV'}

pos = dict(colerless='ADJ',ideas='N',sleep='V', furiously='ADV')

字典的键必须是不可改变的类型，如字符串和元组。如果尝试使用可变键(如链表)定义字典会得到一个TypeError。

**默认字典**

如果试图访问一个不在字典中的键，会得到一个错误。然而，如果一个字典能为这个新键自动创建一个条目并给它一个默认值，如 0 或者一个空链表，将是有用的。自从 Python 2.5 以来，一种特殊的称为 defaultdict 的字典已经出现。为了使用它，必须提供一个参数，用来创建默认值，如:int、float、str 、list 、dict、tuple：

frequency = nltk.defaultdict(int)

print(frequency['ideas'])

pos = nltk.defaultdict(list)

print(pos['ideas'])

这里默认值相当于int()、list()

除了指定字典项的默认值为一个特定的数据类型的默认值之外，也可以指定任何我们喜欢的默认值，只要提供可以无参数的被调用产生所需值的函数的名字，如创建一个任一条目的默认值是'N'的字典：

pos = nltk.defaultdict(lambda:'N')

print(pos['blog'])

print(list(pos.items()))

处理文本中只出现过一次的词,预处理一个文本，在一个默认字典的帮助下，替换低频词汇为一个特殊的“超出词汇表”标识符，UNK(out of vocabulary)。创建一个默认字典，映射每个词为它们的替换词。最频繁的 n 个词将被映射到它们自己。其他的被映射到 UNK:

alice = nltk.corpus.gutenberg.words('carroll-alice.txt')

vocab = nltk.FreqDist(alice)

v1000 = [word for (word,\_) in vocab.most\_common(1000)]

mapping = nltk.defaultdict(lambda:'UNK')

for v in v1000:

mapping[v] = v

alice2 = [mapping[v] for v in alice]

print(alice2[:50])

print(len(set(alice2)))

**递增地更新字典**

可以使用字典计数出现的次数。首先初始化一个空的 defaultdict，然后处理文本中每个词性标记。如果标记以前没有见过，就默认计数为零。每次我们遇到一个标记，就使用+=运算符递增它的计数：

counts = nltk.defaultdict(int)

for (word,tag) in brown.tagged\_words(categories='news',tagset='universal'):

counts[tag] += 1

print(counts['NOUN'])

print(sorted(counts)[:20])

from operator import itemgetter

print(sorted(counts.items(),key=itemgetter(1),reverse=True))

print([t for t,c in sorted(counts.items(),key=itemgetter(1),reverse=True)])

该列表演示了一个重要的按值排序一个字典的习惯用法，按频率递减顺序显示词汇。sorted()的第一个参数是要排序的项目，它是由一个 POS 标记和一个频率组成的元组的链表。第二个参数使用函数 itemgetter()指定排序键。在一般情况下，itemgetter(n) 返回一个函数，这个函数可以在一些其他序列对象上被调用获得这个序列的第 n 个元素。

pair[1] = itemgetter(1)(pair)

sorted()的最后一个参数指定项目是否应被按相反的顺序返回。Sorted函数默认由低到高排序，reverse=True使其由高到低排序，即频率值递减

该例还包括第二个有用的习惯用法，初始化一个 defaultdict，然后使用 for 循环来更新其值。另一个实例，按最后两个字母索引词汇:

last\_letters = nltk.defaultdict(list)

words = nltk.corpus.words.words('en')

for word in words:

key = word[-2:]

last\_letters[key].append(word)

print(last\_letters['ly'][:20])

print(last\_letters['zy'][:20])

使用相同的模式创建一个颠倒顺序的词字典:

anagrams = nltk.defaultdict(list)

for word in words:

key = ''.join(sorted(word))

anagrams[key].append(word)

print(anagrams['aeilnrt'])

即所有包含索引中的字母但顺序任意排列的词。

由于积累这样的词是如此常用的任务，NLTK以 nltk.Index()的形式提供一个创建 defaultdict(list)更方便的方式：

anagrams = nltk.Index((''.join(sorted(w)),w) for w in words)

print(anagrams['aeilnrt'])

nltk.Index 是一个额外支持初始化的 defaultdict(list)。类似的，nltk.FreqDist 本质上是一个额外支持初始化的 defaultdict(int)(附带排序和绘图方法)。

**复杂的键和值**

研究一个词可能的标记的范围，给定词本身和它前一个词的标记:

pos = nltk.defaultdict(lambda:nltk.defaultdict(int))

brown\_news\_tagged = brown.tagged\_words(categories='news',tagset='universal')

for ((w1,t1),(w2,t2)) in nltk.bigrams(brown\_news\_tagged):

pos[(t1,w2)][t2]+=1

print(pos[('DET','right')])

这个例子使用一个字典，它的条目的默认值也是一个字典(其默认值是 int()，即 0)。

注意如何遍历已标注语料库的双连词，每次遍历处理一个词-标记对。每次通过循环时，更新字典 pos 中的条目 (t1, w2)，一个标记和它后面的词。当我们在 pos 中查找一个项目时，必须指定一个复合键，然后得到一个字典对象。一个 POS 标注器可以使用这些信息来决定词 right，前面是一个限定词时，应标注为 ADJ。

**颠倒字典**

字典支持高效查找，只要你想获得任意键的值。如果 d 是一个字典，k 是一个键，输入 d[K]，就立即获得值。给定一个值查找对应的键要慢一些和麻烦一些:

counts = nltk.defaultdict(int)

for word in nltk.corpus.gutenberg.words('milton-paradise.txt'):

counts[word]+=1

print([key for (key,value) in counts.items() if value == 32])

如果希望经常做这样的一种“反向查找”，建立一个映射值到键的字典是有用的。

在**没有两个键具有相同的值**的情况下，这是一个容易的事。只要得到字典中的所有键-值对，并 创建一个新的值-键对字典：

pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}

pos2 = dict((value,key) for (key,value) in pos.items())

print(pos2['N'])

接下来将我们的词性字典做的更实用些，使用字典的 update()方法加入再一些词到 pos 中，创建多个键具有相同的值的情况。这样一来，刚才看到的反向查找技术就将不再起作用(因为字典中一个键只能对应一个值，值-键对发生覆盖)。作为替代，不得不使用 append()积累词和每个词性，如下所示:

pos.update({'cats': 'N','scratch': 'V', 'peacefully': 'ADV', 'old': 'ADJ'})

pos2 = nltk.defaultdict(list)

for key,value in pos.items():

pos2[value].append(key)

print(pos2['ADV'])

此时所有的值均为链表形式

可以使用NLTK 中的索引支持更容易的做同样的事，如下所示:

pos2 = nltk.Index((value,key) for (key,value) in pos.items())

print(pos2['ADV'])

Python字典方法:常用的方法与字典相关习惯用法的总结

示例 说明

d = {} 创建一个空的字典，并将分配给d

d[key] = value 分配一个值给一个给定的字典键

d.keys() 字典的键的链表

list(d) 字典的键的链表

sorted(d) 字典的键，排序

key in d 测试一个特定的键是否在字典中

for key in d 遍历字典的键

d.values() 字典中的值的链表

dict([(k1,v1), (k2,v2), ...]) 从一个键-值对链表创建一个字典

d1.update(d2) 添加 d2 中所有项目到 d1

defaultdict(int) 一个默认值为 0 的字典

**5.4 自动标注**

一个词的标记依赖于这个词和它在句子中的上下文。出于这个原因，我们将处理(已标注)句子层次而不是词汇层次的数据。

**默认标注器**

最简单的标注器是为每个标识符分配同样的标记,它建立了标注器性能的一个重要的底线。用最有可能的标记标注每个词。找出哪个标记是最有可能的(现在使用未简化标记集):

tags = [tag for (word,tag) in brown.tagged\_words(categories='news')]

print(nltk.FreqDist(tags).max())

创建一个将所有词都标注成 NN 的标注器:

raw = 'I do not like green eggs and ham, I do not like them Sam I am!'

tokens = nltk.word\_tokenize(raw)

default\_tagger = nltk.DefaultTagger('NN')

print(default\_tagger.tag(tokens))

评估这种方法的表现。在一个典型的语料库中，它只标注正确了八分之一的标识符:

print(default\_tagger.evaluate(brown\_tagged\_sents))

默认的标注器给每一个单独的词分配标记，即使是之前从未遇到过的词。碰巧的是，一旦我们处理了几千词的英文文本之后，大多数新词都将是名词。这意味着，默认标注器可以帮助我们提高语言处理系统的稳定性。

**正则表达式标注器**

正则表达式标注器**基于匹配模式**分配标记给标识符。例如:我们可能会猜测任一以 ed 结尾的词都是动词过去分词，任一以's 结尾的词都是名词所有格。可以用一个正则表达式的列表表示这些:

patterns = [

(r'.\*ing$','VBG'),

(r'.\*ed$','VBD'),

(r'.\*es$','VBZ'),

(r'.\*ould$','MD'),

(r'.\*\'s$','NN$'),

(r'.\*s$','NNS'),

(r'^-?[0-9]+(.[0-9]+)?$','CD'),

(r'.\*','NN')

]

注意，这些是顺序处理的，第一个匹配上的会被使用。现在我们可以建立一个标注器，并用它来标记一个句子。做完这一步会有约五分之一是正确的:

regexp\_tagger = nltk.RegexpTagger(patterns)

print(regexp\_tagger.tag(brown\_sents[3])[:20])

print(regexp\_tagger.evaluate(brown\_tagged\_sents))

最终的正则表达式«.\*»是全面捕捉的，标注所有词为名词。除了作为正则表达式

标注器的一部分重新指定这个，这与默认标注器是等效的(只是效率低得多)。

**查询标注器**

很多高频词没有 NN 标记。找出 100 个最频繁的词，存储它们最有可能的标记。然后可以使用这个信息作为“查找标注器”(NLTK UnigramTagger)的模型:

fd = nltk.FreqDist(brown.words(categories='news'))

cfd = nltk.ConditionalFreqDist(brown.tagged\_words(categories='news'))

most\_freq\_words = fd.most\_common(100)

likely\_tags = dict((word,cfd[word].max()) for (word,\_) in most\_freq\_words)

baseline\_tagger = nltk.UnigramTagger(model=likely\_tags)

print(baseline\_tagger.evaluate(brown\_tagged\_sents))

仅仅知道 100 个最频繁的词的标记就能正确标注很大一部分标识符(近一半，事实上)。来看看它在一些未标注的输入文本上做的如何:

sent = brown.sents(categories='news')[3]

print(baseline\_tagger.tag(sent))

许多词都被分配了一个 None 标签，因为它们不在 100 个最频繁的词之中。在这些情况下，我们想分配默认标记 NN。换句话说，我们要先使用查找表，如果它不能指定一个标记就使用默认标注器，这个过程叫做**回退(backoff)**。我们可以做到这个，通过指定一个标注器作为另一个标注器的参数，如下所示。现在查找标注器将只存储名词以外的词的词-标记对，只要它不能给一个词分配标记，它将会调用默认标注器:

baseline\_tagger = nltk.UnigramTagger(model=likely\_tags,

backoff=nltk.DefaultTagger('NN'))

**评估**

在前面的例子中，一直在强调准确性得分。事实上，这些工具的性能评估是 N LP 的一个中心主题。一个模块输出中的任何错误都在下游模块大大的放大。

我们对比专家分配的标记来评估一个标注器的性能。由于我们通常很难获得专业和公正的人的判断，所以使用**黄金标准**测试数据来代替。这是一个已经手动标注并作为自动系统评 估标准而被接受的语料库。当标注器对给定词猜测的标记与黄金标准标记相同，标注器被视 为是正确的。

**5.5 N-gram 标注**

**一元标注( Unigram Tagging)**

一元标注器基于一个简单的统计算法:对每个标识符分配这个独特的标识符最有可能的标记。一个一元标注器的行为就像一个查找标注器，除了有一个更方便的建立它的技术，称为**训练**。训练一个一元标注器，用它来标注一个句子，然后评估:

brown\_tagged\_sents = brown.tagged\_sents = brown.tagged\_sents(categories='news')

brown\_sents = brown.sents(categories='news')

unigram\_tagger = nltk.UnigramTagger(brown\_tagged\_sents)

print(unigram\_tagger.tag(brown\_sents[2007]))

print(unigram\_tagger.evaluate(brown\_tagged\_sents))

我们**训练**一个 UnigramTagger，通过在我们初始化标注器时指定已标注的句子数据作为参数。训练过程中涉及检查每个词的标记，将所有词的最可能的标记存储在一个字典里面，这个字典存储在标注器内部。

**分离训练和测试数据**

在一些数据上训练一个标注器时，必须小心不要在相同的数据上测试。一个只是记忆它的训练数据，而不试图建立一个一般的模型的标注器会得到一个完美的得分，但在标注新的文本时将是无用的。

相反，应该分割数据，90%为训练数据，其余 10%为测试数据:

size = int(len(brown\_tagged\_sents)\*0.9)

print(size)

train\_sents = brown\_tagged\_sents[:size]

test\_sents = brown\_tagged\_sents[size:]

unigram\_tagger = nltk.UnigramTagger(train\_sents)

print(unigram\_tagger.evaluate(test\_sents))

**一般的 N-gram 的标注**

在基于 unigrams 处理一个语言处理任务时，使用上下文中的一个项目。标注的时候，只考虑当前的标识符，与更大的上下文隔离。给定一个模型，能做的最好的是为每个词标注其先验的最可能的标记。

一个 **n-gram 标注器**是一个 unigram 标注器的一般化，它的上下文是**当前词**和它前面 n- 1 个标识符的**词性标记**。一个 n-gram 标注器挑选在给定的上下文中最有可能的标记。

1-gram 标注器是一元标注器(unigram tagger)另一个名称:即用于标注一个标识符的上下文的只是标识符本身。2-gram 标注器也称为二元标注器 (bigram taggers)，3-gram 标注器也称为三元标注器(trigram taggers)。

NgramTagger类使用一个已标注的训练语料库来确定对每个上下文哪个词性标记最 有可能。bigram 标注器：

bigram\_tagger = nltk.BigramTagger(train\_sents)

print(bigram\_tagger.tag(brown\_sents[2007]))

unseen\_sent = brown\_sents[4203]

print(bigram\_tagger.tag(unseen\_sent))

bigram 标注器能够标注训练中它看到过的句子中的所有词，但对一个没见过的句子表现很差。只要遇到一个新词(如 13.5)，就无法给它分配标记。它不能标注下面的词(如:million)，即使是在训练过程中看到过的，只是因为在训练过程中从来没有见过它前面有一个 None 标记的词。因此，标注器标注句子的其余部分也失败了。它的整体准确度得分非常低:

print(bigram\_tagger.evaluate(test\_sents))

当 n 越大，上下文的特异性就会增加，我们要标注的数据中包含训练数据中不存在的上下文的几率也增大。这被称为数据稀疏问题，在 NLP 中是相当普遍的。因此，我们的研究结果的精度和覆盖范围之间需要有一个权衡(这与信息检索中的**精度/召回权衡**有关)。

N-gram 标注器不应考虑跨越句子边界的上下文。因此， NLTK的标注器被设计用于句子链表，一个句子是一个词链表。在一个句子的开始，tn-1 和前面的标记被设置为 None。

**组合标注器**

更精确的算法在很多时候落后于具有更广覆盖范围的算法。

组合 bigram 标注器、unigram 标注器和一个默认标注器:

1. 尝试使用 bigram 标注器标注标识符。

2. 如果 bigram 标注器无法找到一个标记，尝试 unigram 标注器。

3. 如果 unigram 标注器也无法找到一个标记，使用默认标注器。

大多数 NLTK 标注器允许指定一个**回退标注器**。回退标注器自身可能也有一个回退标注器:

t0 = nltk.DefaultTagger('NN')

t1 = nltk.UnigramTagger(train\_sents,backoff=t0)

t2 = nltk.BigramTagger(train\_sents,backoff=t1)

print(t2.evaluate(test\_sents))

t3 = nltk.TrigramTagger(train\_sents,backoff=t2)

print(t3.evaluate(test\_sents))

在标注器初始化时指定回退标注器，从而使训练能利用回退标注器。

如果在一个上下文中 bigram 标注器将分配与它的 unigram 回退标注器一样的标记，那么 bigram 标注器丢弃训练的实例。这样保持尽可能小的 bigram 标注器模型。可以进一步指定一个标注器需要看到一个上下文的多个实例才能保留它。例如:nltk.Bigram Tagger(sents, cutoff=2, backoff=t1)将会丢弃那些只看到一次或两次的上下文。(不一定会提高准确率)

**标注生词**

一个有用的基于上下文标注生词的方法是限制一个标注器的词汇表为最频繁的 n 个词，使用 5.3 节中的方法替代每个其他的词为一个特殊的词 UNK。训练时，一个 unigram 标注器可能会学到 UNK 通常是一个名词。然而，n-gram 标注器会检测它的一些其他标记中的上下文。例如:如果前面的词是 to(标注为 TO)，那么 UNK 可能会被标注为一个动词。

**存储标注器**

保存标注器 t2 到文件 t2.pkl:

>>> from pickle import dump

>>> output = open('t2.pkl','wb')

>>> dump(t2,output,-1)

>>> output.close()

在一个单独的 Python 进程中载入保存的标注器:

from pickle import load

input = open('t2.pkl','rb')

tagger = load(input)

input.close()

检查它是否可以用来标注:

text = """The board's action shows what free enterprise

is up against in our complex maze of regulatory laws ."""

tokens = text.split()

print(tagger.tag(tokens))

**性能限制**

一个 trigram 标注器的情况可能遇到多少词性歧义的情况：

cfd = nltk.ConditionalFreqDist(

((x[1],y[1],z[0]),z[1])

for sent in brown\_tagged\_sents

for x,y,z in nltk.trigrams(sent))

ambiguous\_contexts = [c for c in cfd.conditions() if len(cfd[c])>1]

print(sum(cfd[c].N() for c in ambiguous\_contexts)/cfd.N())

因此，1/20 的 trigrams 是有歧义的。给定当前单词及其前两个标记，根据训练数据，在5%的情况中，有一个以上的标记可能合理地分配给当前词。假设我们总是挑选在这种含糊不清的上下文中最有可能的标记，可以得出 trigram 标注器性能的一个下界。

调查标注器性能的另一种方法是研究它的错误。有些标记可能会比别的更难分配，可能需要专门对这些数据进行预处理或后处理。一个方便的方式查看标注错误是**混淆矩阵**。它用图表表示期望的标记(黄金标准)与实际由标注器产生的标记:

test\_tags = [tag for sent in brown.sents(categories='editorial')

for (word,tag) in t2.tag(sent)]

gold\_tags = [tag for (word,tag) in brown.tagged\_words(categories='editorial')]

print(nltk.ConfusionMatrix(gold\_tags,test\_tags))

或许标记之间很难做出的区分可以被丢弃，因为它在一些较大的处理任务的上下文中并不重要。

分析标注器性能界限的另一种方式来自人类标注者之间并非 100%的意见一致。

一般情况下，标注过程会消除区别:例如:当所有的人称代词被标注为 PRP 时，词的特性通常会失去。与此同时，标注过程引入了新的区别从而去除了含糊之处:例如:deal 标注为 VB 或 NN。这种消除某些区别并引入新的区别的特点是标注的一个重要的特征，有利于分类和预测。

训练数据中的歧义导致标注器性能的上限。

**跨句子边界标注**

使用已标注句子的链表来训练、运行和评估标注器，避免前面句子的最后一个词和句子结尾的标点符号这些通常无关的部分带来负面的影响

**5.6 基于转换的标注**

在本节中，我们考察 Brill 标注，一种归纳标注方法，它的性能很好，使用的模型只有 n-gram 标注器的很小一部分。

Brill 标注是一种基于转换的学习，以它的发明者命名。一般的想法很简单:猜每个词的标记，然后返回和修复错误的。与 n-gram 标注一样，这是有监督的学习方法，因为我们需要已标注的训练数据来评估标注器的猜测是否是一个错误。然而，不像 n-gram 标注，它不计数观察结果，只编制一个**转换修正规则链表**。

Brill 标注的想法:以大笔画开始，然后修复细节，一点点的细致的改变。

首先使用 unigram 标注器标注， 然后运用规则修正错误。

所有规则由以下形式的模板产生:“在上下文 C 中替换 T1 为 T2。”典型的上下文是之前或之后的词的内容或标记，或者当前词的两到三个词范围内出现的一个特定标记。在其训练阶段，T1，T2 和 C 的标注器猜测值创造出数以千计的候选规则。每一条规则都根据其净收益打分:它修正的不正确标记的数目减去它错误修改的正确标记的数目。

规则是语言学可解释的。

**5.7 如何确定一个词的分类**

**形态学线索**

一个词的内部结构可能为这个词分类提供有用的线索。比如各种后缀。

**句法线索**

另一个信息来源是一个词可能出现的典型的上下文语境。

**语义线索**

最后，一个词的意思对其词汇范畴是一个有用的线索。在现代语言学，词类的语义标准受到怀疑，主要是因为它们很难规范化。然而，语义标准巩固了我们对许多词类的直觉，使我们能够在不熟悉的语言中很好的猜测词的分类。

**新词**

所有的语言都学习新的词汇。新晋产生的所有新词都是名词，这反映在名词被称为**开放类**。相反，介词被认为是一个**封闭类**。也就是说，只有有限的词属于这个类别，词类成员随着很长时间的推移才逐渐改变。

**词性标记集中的形态学**

普通标记集经常捕捉一些**构词**信息，即词借助它们的句法角色获得的一种形态标记的信息。

更细粒度的标记集提供有关这些不同语法形式的有用信息，可以帮助尝试检测标记序列模式的其它处理者。

布朗标记集的一些构词区别

形式 类别 标记

go 基本 VB

goes 第三人称单数 VBZ

gone 过去分词 VBN

going 动名词 VBG

went 一般过去时 VBD

总的来说，这种动词细粒度标记意味着使用此标记集的自动标注器能有效开展有限数量的形态分析。

多数词性标注集使用相同的基本类别，如名词、动词、形容词和介词。然而，标记集的相互区别不仅在于它们如何细致的将词分类，也在于它们如何界定其类别。这种标记集的变化是不可避免的，因为词性标记被以不同的方式用于不同的任务。换句话说，没有一个“正确的方式”来分配标记，只有根据目标不同或多或少有用的方法。