PYTHON自然语言处理（中文版）

PYTHON自然语言处理（中文版）

一、语言处理与Python

1、NLTK入门

下载数据集：输入命令：`nltk.download()`， 之后选择要下载的数据集（如book）

若出现拒接连接则将https://github.com/nltk/nltk_data/tree/gh-pages/packages文件全部复制替换到C:\Users[用户名]\AppData\Roaming\nltk_data下即可解决

查看模块内容：`from nltk.book import *`

检索指定词：`text1.concordance("monstrous")`

检索指定词的上下文：`text2.similar("life")`

检索多个词共同的上下文：`text2.common_contexts(["love", "dear"])`

计算指定词从开头算起其前面有多少词：`text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])`

随机生成指定文本风格的文字：`text3.generate()`

计算特定词出现的次数:`text3.count("smote")`

获取指定文本的词频信息（字典）：`fdist1=FreqDist(text1)`

检索只出现一次的词：`fdist1.hapaxes()`

前50高频词累计频率图（cumulative=False是频率分布图）：`fdist1.plot(50, cumulative=True)`

获取相邻词对：`bigrams(['more', 'is', 'than', 'done'])`

检索高频双连词：`text4.collocations()`

检索指定长度的词的数量：`fdist.items()`

计算词频：`fdist.freq('I')`

创建频率分布表：`fdist.tabulate()`

2、自然语言处理

文本对齐：给出一个双语文档，可以自动配对组成句子的过程

二、获得文本语料和词汇资料

1、单语料库使用

书籍

```python
from nltk.corpus import gutenberg  # 书籍文本库
gutenberg.fileids()  # 获取语料库标识符

emma = gutenberg.words('austen-emma.txt')  # 获取《爱玛》语料
emma = nltk.Text(emma)  # 获取单个文本检索信息对象
emma.concordance('surprize')

gutenberg.raw('austen-emma.txt')  # 原文
gutenberg.words('austen-emma.txt')  # 词汇集
gutenberg.sents('austen-emma.txt')  # 句子（词链表）
```

网络文本

```python
from nltk.corpus import webtext  # 网络文本集
for fileid in webtext.fileids():
    print(fileid, webtext.raw(fileid)[:50])
```

即时聊天会话语料库

10-19-20s_706posts.xml表示包括2006年10月19日从20多岁聊天室收集的706个帖子。

```python
from nltk.corpus import nps_chat  # 即时聊天回话预料库
chatroom = nps_chat.posts('10-19-20s_706posts.xml')
chatroom[123]
```

布朗语料库

ID	文件	文体	描述
A16	ca16	新闻 news	Chicago Tribune: Society Reportage
B02	cb02	社论 editorial	Christian Science Monitor: Editorials
C17	cc17	评论 reviews	Time Magazine: Reviews
D12	cd12	宗教 religion	Underwood: Probing the Ethics of Realtors
E36	ce36	爱好 hobbies	Norling: Renting a Car in Europe
F25	cf25	传说 lore	Boroff: Jewish Teenage Culture
G22	cg22	纯文学 belles_lettres	Reiner: Coping with Runaway Technology
H15	ch15	政府 government	US Office of Civil and Defence Mobilization: The Family Fallout Shelter
J17	cj19	博览 learned	Mosteller: Probability with Statistical Applications
K04	ck04	小说 fiction	W.E.B. Du Bois: Worlds of Color
L13	cl13	推理小说 mystery	Hitchens: Footsteps in the Night
M01	cm01	科幻 science_fiction	Heinlein: Stranger in a Strange Land
N14	cn15	探险 adventure	Field: Rattlesnake Ridge
P12	cp12	言情 romance	Callaghan: A Passion in Rome
R06	cr06	幽默 humor	Thurber: The Future, If Any, of Comedy

```
from nltk.corpus import brown  # 英语电子预料库
brown.categories()

brown.words(categories='news')

brown.words(fileids=['cg22'])

brown.sents(categories=['news', 'editorial', 'reviews'])
```

比较不同文体中情态动词的用法:

```
from nltk.corpus import brown

news_text = brown.words(categories='news')
fdist = nltk.FreqDist([w.lower() for w in news_text])  # 统计词频
modals = ['can', 'could', 'may', 'might', 'must', 'will']
# 显示指定文体的情态动词词频信息
for m in modals:
    print(m + ':' + str(fdist[m]))

# 获取不同文体以及对应的词
cfd = nltk.ConditionalFreqDist(
    (genre, word) for genre in brown.categories()
                  for word in brown.words(categories=genre))

genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance',
'humor']
modals = ['can', 'could', 'may', 'might', 'must', 'will']
# 对指定文体的指定词计算词频
cfd.tabulate(conditions=genres, samples=modals)
```

路透社语料库

```
from nltk.corpus import reuters  # 路透社预料库
reuters.fileids()[:10]

reuters.categories()[:20]

reuters.categories(['training/9865', 'test/14826'])  # 获取指定集的类别信息

reuters.fileids(['barley', 'corn'])[:10]  # 指定类别查询语料集

reuters.words('training/9865')[:15]  # 开头大写的是题目

reuters.words(['training/9865', 'training/9880'])[:15]

reuters.words(categories=['barley'])[:15]
```

就职演说语料库

```python
from nltk.corpus import inaugural  # 就职演说预料库
inaugural.fileids()[:10]

[fileid[:4] for fileid in inaugural.fileids()][:10]  # 获取时间

# 绘制不同词在随时间演讲时的变换
cfd = nltk.ConditionalFreqDist(
    (target, fileid[:4]) for fileid in inaugural.fileids()
                              for w in inaugural.words(fileid)
                                  for target in ['america', 'citizen']
                                      if w.lower().startswith(target))
cfd.plot()
```

标注文本语料库：https://www.nltk.org/howto/

词汇列表语料库

```python
# 过滤高频词汇
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())  # 不重复英文单词
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())  # 所有不重复单词
    # 返回text_vocab不同于english_vocab的词
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)

unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))[:10]
```

```python
from nltk.corpus import stopwords  # 停用词
stopwords.words('english')[:10]
```

其他语言语料库

```python
from nltk.corpus import udhr  # 引入世界人权宣言语料

udhr.fileids()[:20]  # 检索语言

# 绘制不同语言在《世界人权宣言》的字长差异
languages = ['Chickasaw', 'English', 'German_Deutsch']
cfd = nltk.ConditionalFreqDist(
    (lang, len(word)) for lang in languages
                          for word in udhr.words(lang + '-Latin1'))
cfd.conditions()  # 查看条件,对于每个cfd['xxx']都是一个频率分布
cfd.plot(cumulative=True)
```

2、使用自己的预料库

```python
from nltk.corpus import PlaintextCorpusReader
corpus_root = '.'  # 设置路径
wordlists = PlaintextCorpusReader(corpus_root, ".*")  # 查询
wordlists.fileids()  # 查看文件夹内容

wordlists.words('my_text.txt')  # 使用
```

3、生成随机文本

```
def generate_model(cfdist, word, num=15):  # 生成随机文本
    for i in range(num):
        print(word)
        word = cfdist[word].max()

text = nltk.corpus.genesis.words('english-kjv.txt')
bigrams = nltk.bigrams(text)  # 相邻词对
cfd = nltk.ConditionalFreqDist(bigrams)

cfd['living']  # 查看'living'后面可能跟哪些词

generate_model(cfd, 'living')
```

4、条件概率

```
pairs = [('a', 'b'), ('b', 'c'), ('c', 'a'), ('c', 'd')]
cfd = nltk.ConditionalFreqDist(pairs)
cfd  # 有3个条件

cfd.conditions()  # 将条件按字母排序

cfd['c']  # 查看此条件下的频率分布

cfd['c']['b']  # 查看指定条件和样本的频率

cfd.tabulate()  # 为条件频率分布制表

cfd.plot()  # 为条件频率分布绘图

# 获取不大于指定词频的单词，即得到词的词频都<=target的词频
target = 'abcde'
wordlist = nltk.corpus.words.words()
for w in wordlist:
    if nltk.FreqDist(w) <= nltk.FreqDist(target):
        print(w)
```

5、词典语料

```
names = nltk.corpus.names  # 名字预料库
names.fileids()

male_names = names.words('male.txt')
male_names[:10]

# ================================================================
entries = nltk.corpus.cmudict.entries()  # 发音语料

for entry in entries[:10]:
    print(entry)  # 每个词后面是声音的标签，类似音节（同一个词可能多种发音）

# 找到所有发音开头与aalseth相似的词汇
syllable = 'AA1'
[word for word, pron in entries if syllable in pron][:10]
```

```

def stress(pron):
    return [char for phone in pron for char in phone if char.isdigit()]

# 提取重音词，其中主重音（1）、次重音（2）和无重音（0）
[w for w, pron in entries if stress(pron) == ['0', '1', '2', '0']]

# 找到词汇的最小受限集合
p3 = [(pron[0] + '-' + pron[2], word) for (word, pron) in entries
     if pron[0] == 'P' and len(pron) == 3]

cfd = nltk.ConditionalFreqDist(p3)
for template in cfd.conditions():  # 获取所有条件，如'P-TH'、'P-K'等
    if len(cfd[template]) > 0:  # 对应条件有元素
        words = cfd[template]
        wordlist = ' '.join(words)
        print(template, wordlist[:70] + ' ...')

# 获取发音词典（若无则可以手动添加，但是对NLTK预料库不影响）
prondict = nltk.corpus.cmudict.dict()
prondict['fire']

# ===========================================================================
from nltk.corpus import swadesh  # 比较词表，多种语言都含200左右常用词
swadesh.fileids()

swadesh.words('en')[:10]  # 英语

fr2en = swadesh.entries(['fr', 'en'])  # 不同语言的同源词
fr2en[:10]

translate = dict(fr2en)  # 翻译 fr -> en
translate['chien']

# 更新翻译词典
de2en = swadesh.entries(['de', 'en'])
translate.update(dict(de2en))  # 加入德语
translate['Hund']
```

6、WordNet

面向语义的英语词典，具有丰富的结构。下述的上位词和下位词都是同义词之间的空间关系。

同义词集：形容对某一事物的特征

同义词：相同特征的不同描述语

词条：同义词集和词的配对

```

from nltk.corpus import wordnet as wn

wn.synsets('motorcar')  # 查找'motorcar'的同义词集
wn.synset('car.n.01').lemma_names()  # 查看同义词
wn.synset('car.n.01').definition()  # 定义
wn.synset('car.n.01').examples()  # 例句
wn.synset('car.n.01').lemmas()  # 同义词集的所有词条
wn.lemmas('car')  # 查看包含'car'的所有词条
wn.lemma('car.n.01.automobile')  # 查找特定的词条
```

```
wn.lemma('car.n.01.automobile').synset()  # 得到词条的对应同义词集
wn.lemma('car.n.01.automobile').name()  # 得到词条的"名字"

wn.synsets('car')  # 与'motorcar'不同，有5个词集
for synset in wn.synsets('car'):  # 查看每个同义词集的同义词
    print(synset.lemma_names())
```

```
motorcar = wn.synset('car.n.01')
type_of_motorcar = motorcar.hyponyms()  # 下位词（包含的词集）

motorcar.hypernyms()  # 上位词（父词集）

paths = motorcar.hypernym_paths()  # 上位词路径

motorcar.root_hypernyms()  # 根上位词
```

```
wn.synset('tree.n.01').part_meronyms()  # 包含的部分
wn.synset('tree.n.01').substance_meronyms()  # 其实质
wn.synset('tree.n.01').member_holonyms()  # 组成的整体

for synset in wn.synsets('mint', wn.NOUN):
    print(synset.name(), ':', synset.definition())
wn.synset('mint.n.04').part_holonyms()  # 叶子是薄荷的一部分
wn.synset('mint.n.04').substance_holonyms()  # 用薄荷油制作的糖材质是薄荷

wn.synset('walk.v.01').entailments()  # 走路的"需求"包括抬脚
wn.synset('eat.v.01').entailments()
wn.lemma('supply.n.02.supply').antonyms()  # "供给"的反义词是"需求"
dir(wn.synset('harmony.n.02'))  # 查看指定词集的所有方法
```

```
right = wn.synset('right_whale.n.01')
orca = wn.synset('orca.n.01')
minke = wn.synset('minke_whale.n.01')
tortoise = wn.synset('tortoise.n.01')
novel = wn.synset('novel.n.01')
right.lowest_common_hypernyms(minke)  # 求语义最相近的

wn.synset('baleen_whale.n.01').min_depth()  # 同义词集深度
wn.synset('whale.n.02').min_depth()
wn.synset('entity.n.01').min_depth()

right.path_similarity(minke)  # 求取相似度

help(wn)
```

三、加工原料文本

1、从网络和硬盘访问文本

访问本地文本可以直接使用`open('xxx').read()`来获取raw

```
from urllib.request import urlopen

```

```
url = "http://www.gutenberg.org/files/2554/2554-0.txt"
raw = urlopen(url).read()
raw = str(raw)

tokens = nltk.word_tokenize(raw)  # 生成token列表
nltk.tokenwrap(raw)  # 生成token字符串

text = nltk.Text(tokens)
text.collocations()  # 检测高频双连词

raw.find("PART I")
raw.rfind("the subject of a new story")

raw = raw[5866: 1338204]
raw.find("PART I")

raw = raw[5866: 1338204]
raw.find("the subject of a new story")

'''
需要清除html情况
'''
url = "http://www.gutenberg.org/files/2554/2554-h/2554-h.htm"
html = urlopen(url).read()

from bs4 import BeautifulSoup

raw = BeautifulSoup(html).get_text()
tokens = nltk.word_tokenize(raw)

text = nltk.Text(tokens)
text.concordance('forbidden')  # 检索指定词

'''
读取NLTK语料库文件
'''
path = nltk.data.find('corpora/gutenberg/melville-moby_dick.txt')
raw = open(path, 'r').read()
```

2、编码

解码：翻译为Unicode

编码：将Unicode转化为其他编码的过程

文件开头添加：`# -*- coding: utf-8 -*-`

3、正则表达式

^：表示以后面的字符为开头，若放在方括号里则代表除了括号内的字符之外

$：表示以前面的字符为结尾

.：匹配任意单个字符

+：匹配至少一个字符，也被成为闭包

*：匹配至少零个字符，也被成为闭包

\：表示后面的一个字符不具备特殊匹配含义

{a, b}：表示前面的项目指定重复次数

字符串前面加'r'表示原始字符串

符号	功能
\b	词边界（零宽度）
\d	任一十进制数字（相当于[0-9]）
\D	任何非数字字符（等价于[^ 0-9]）
\s	任何空白字符（相当于[\t\n\r\f\v]）
\S	任何非空白字符（相当于[^ \t\n\r\f\v]）
\w	任何字母数字字符（相当于[a-zA-Z0-9_]）
\W	任何非字母数字字符（相当于[^a-zA-Z0-9_]）
\t	制表符
\n	换行符

```
import re

wordlist = [w for w in nltk.corpus.words.words() if w.islower()]
[w for w in wordlist if re.search('ed$', w)][:10]  # 查找以ed结尾的词

[w for w in wordlist if re.search('^..j..t..$', w)][:10]  # 查找指定位置的词

[w for w in wordlist if re.search('^a(b|c)k', w)][:10]

word = 'kfbhfuicbhflchbirlblirb'
re.findall(r'[aeiou]', word)

[int(n) for n in re.findall(r'[0-9]{2,4}', '2019-12-31')]
```

```
# 从罗托卡特语词汇中提取所有辅音-元音序列
cv_word_pairs = [(cv, w) for w in nltk.corpus.toolbox.words('rotokas.dic')
                         for cv in re.findall(r'[ptksvr][aeiou]', w)]
cv_index = nltk.Index(cv_word_pairs)  # 转换为索引表

cv_index['po']
```

搜索已分词文本

```
from nltk.corpus import gutenberg, nps_chat
moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
# 尖括号用于标记标识符的边界，
# 尖括号之间的所有空白都被忽略（这只对nltk中的findall()方法有效）
moby.findall(r'<a><man>')
moby.findall(r'<a><.*><man>')
moby.findall(r'<a>(<.*>)<man>')

w = nps_chat.words()
# 找出以'bro'结尾的三个词组成的短语
chat = nltk.Text(w)
chat.findall(r'<.*><.*><bro>')
# 找出以字母'l'开始的三个或更多词组成的序列
chat.findall(r'<l.*>{3,}')
```

```
# 标记字符串指定模式
nltk.re_show(r'^a|b', 'aaasfgsgdhabbbdsdg')

# 提供正则匹配的图形化程序
nltk.app.nemo()
```

4、词干提取器（中文不需要）

词干：词的原形，不包含后缀如ly,s,es等

```
raw = '''
a word or unit of text to be carried over to a new line automatically as
the margin is reached, or to fit around embedded features such as pictures.
'''
tokens = nltk.word_tokenize(raw)

# 两种不同的内置词干提取器
porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()

[porter.stem(t) for t in tokens][:10]
[lancaster.stem(t) for t in tokens][:10]
```

```
# 使用词干提取器索引文本
class IndexedText(object):
    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in enumerate(text))

    def concordance(self, word, width=40):
        key = self._stem(word)
        wc = width // 4  # 前后各多少词
        for i in self._index[key]:  # 目的是对齐输出显示
            lcontext = ' '.join(self._text[i-wc:i])
            rcontext = ' '.join(self._text[i:i+wc])
            ldisplay = '%*s' % (width, lcontext[-width:])
            rdisplay = '%-*s' % (width, rcontext[:width])
            print(ldisplay, rdisplay)

    def _stem(self, word):
        return self._stemmer.stem(word).lower()


poter = nltk.PorterStemmer()
grail = nltk.corpus.webtext.words('grail.txt')
text = IndexedText(porter, grail)
text.concordance('lie')
```

5、词形归并/词形还原（中文不需要）

词形还原：将复杂的词形表示转换为最基本的状态，如women装换为woman，但是不转换普通的过去式等

```
# WordNet词形归并器删除词缀产生的词都是在它的字典中的词
raw = '''
a women or unit of text to be carried lying to a new line automatically as
the margin is reached, or to fit around embedded features such as pictures.
'''
tokens = nltk.word_tokenize(raw)

wnl = nltk.WordNetLemmatizer()
[wnl.lemmatize(t) for t in tokens][:10]
```

6、分词

模拟退火算法：迭代求解策略的一种随机寻优算法。模拟退火算法从某一较高初温出发，伴随温度参数的不断下降，结合概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。模拟退火算法是一种通用的优化算法，理论上算法具有概率的全局优化性能

```
text = 'doyouseethekittyseethedoggydoyoulikethekittylikethedoggy'
seg1 = '0000000000000001000000000010000000000000000010000000000'
seg2 = '0100100100100001001001000010100100010010000100010010000'

# 根据segs对text进行分段/词
def segment(text, segs):
    words = []
    last = 0
    for i in range(len(segs)):
        if segs[i] == '1':
            words.append(text[last: i+1])
            last = i + 1
    words.append(text[last:])
    return words

# 通过合计每个词项与推导表的字符数，作为分词质量的得分。值越小越好
def evaluate(text, segs):
    words = segment(text, segs)
    text_size = len(words)
    lexicon_size = len(' '.join(list(set(words))))  # 查找不重复的所有词
    return text_size + lexicon_size

# 使用模拟退火算法的非确定形搜索
# 一开始仅搜索短语分词：随机扰动1/0；
# 它们与“温度”成正比，每次迭代温度都会降低，扰动边界会减少
from random import randint

def flip(segs, pos):  # 更改pos位置的数字：0到1或1到0
    return segs[:pos] + str(1-int(segs[pos])) + segs[pos+1:]

def flip_n(segs, n):  # 对字符串随机更改n位
    for i in range(n):
        segs = flip(segs, randint(0, len(segs) - 1))
    return segs

def anneal(text, segs, iterations, cooling_rate):
    temperature = float(len(segs))
    while temperature > 0.5:
        best_segs, best = segs, evaluate(text, segs)
```

```
        for i in range(iterations):
            guess = flip_n(segs, int(round(temperature)))  # 产生一个随机更改n位的串
            score = evaluate(text, guess)
            if score < best:
                best, best_segs = score, guess
        score, segs = best, best_segs
        temperature = temperature / cooling_rate  # 更改位数递减
        print(evaluate(text, segs), segment(text, segs))
    print()
    return segs

anneal(text, seg1, 5000, 1.2)
```

四、编写结构化程序

```
# 绘制条形图来统计词频
import nltk

colors = 'rgbcmyk'  # 多种颜色的组合，字符串形式
def bar_chart(categories, words, counts):
    import pylab
    ind = pylab.arange(len(words))
    width = 1 / (len(categories))
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind + c * width, counts[categories[c]],
                         width, color=colors[c % len(colors)])
        bar_groups.append(bars)
    pylab.xticks(ind + width, words)
    pylab.legend([b[0] for b in bar_groups], categories, loc='upper left')
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()

genres = ['news', 'religion', 'hobbies', 'government', 'adventure']
modals = ['can', 'could', 'may', 'might', 'must', 'will']
cfdist = nltk.ConditionalFreqDist((genre, word)
                                  for genre in genres
                                  for word in nltk.corpus.brown.words(categories=genre)
                                  if word in modals)
counts = {}
for genre in genres:
    counts[genre] = [cfdist[genre][word] for word in modals]

bar_chart(genres, modals, counts)
```

```
# 绘制树形图构建上下文关系
import networkx as nx
import matplotlib
from nltk.corpus import wordnet as wn

def traverse(graph, start, node):
    graph.depth[node.name] = node.shortest_path_distance(start)
```

```
    for child in node.hyponyms():
        graph.add_edge(node.name, child.name)
        traverse(graph, start, child)

def hyponym_graph(start):
    G = nx.Graph()
    G.depth = {}
    traverse(G, start, start)
    return G

def graph_draw(graph):
    nx.draw(graph,
            node_size=[16 * graph.degree(n) for n in graph],
            node_color=[graph.depth[n] for n in graph],
            with_labels= False)
    matplotlib.pyplot.show()

dog = wn.synset('cat.n.01')
graph = hyponym_graph(dog)
graph_draw(graph)
```

```
# 奇异值分解
from numpy import linalg
from numpy import array

a = array([[4, 0], [3, -5]])
u, s, vt = linalg.svd(a)
```

五、分类和标注词汇

```
# 使用词性标注器
# CC 并列连词； RB 副词； IN 介词； NN 名词； JJ 形容词 ......
import nltk

text = nltk.word_tokenize("and now for something completely different")
nltk.pos_tag(text)
```

```
# 标注语料库
tagged_token = nltk.tag.str2tuple('fly/NN')
tagged_token
tagged_token[0]

s = 'fly/NN The/AT said/VBD'
[nltk.tag.str2tuple(t) for t in s.split()]

nltk.corpus.brown.tagged_words()
# nltk.corpus.brown.tagged_words(simplify_tags=True)  # 部分语料库使用该方式
nltk.corpus.brown.tagged_words(tagset='universal')

nltk.corpus.nps_chat.tagged_words()
nltk.corpus.sinica_treebank.tagged_words()[:10]  # 中文

from nltk.corpus import brown
brown_news_tagged = brown.tagged_words(categories='news',
tagset='universal')
```

```
tag_fd = nltk.FreqDist(tag for (word, tag) in brown_news_tagged)
tag_fd
tag_fd.plot(cumulative=False)
```

```
# 找出最频繁的名词
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, list(cfd[tag].keys())[:5]) for tag in
cfd.conditions())

tagdict = findtags('NN', nltk.corpus.brown.tagged_words(categories='news'))
for tag in sorted(tagdict):
    print(tag, tagdict[tag])
```

默认标注器

```
brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')

# 查看最多词性是什么，指定为默认
tags = [tag for (word, tag) in brown.tagged_words(categories='news')]
nltk.FreqDist(tags).max()

# 使用
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')
default_tagger.tag(tokens)

# 评估
default_tagger.evaluate(brown_tagged_sents)
```

正则表达式标注器

```
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')
]

regexp_tagger = nltk.RegexpTagger(patterns)
regexp_tagger.tag(brown_sents[3])

regexp_tagger.evaluate(brown_tagged_sents)
```

查询标注器

```
fd = nltk.FreqDist(brown.words(categories='news'))  # 统计词频
```

```
cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))  # 词 - 词性 统计

most_freq_words = list(fd.keys())[:100]  # 最高频的前100词
likely_tags = dict((word, cfd[word].max()) for word in most_freq_words)  # 高频词及高频词性

baseline_tagger = nltk.UnigramTagger(model=likely_tags)

baseline_tagger.evaluate(brown_tagged_sents)

sent = brown.sents(categories='news')[3]
baseline_tagger.tag(sent)  # 不在查询表中时会被标记为空
# 从上可知有些词被标注为空，因为我们需要将未查到的词性标注为默认的
baseline_tagger = nltk.UnigramTagger(model=likely_tags, backoff=nltk.DefaultTagger('NN'))

def performance(cfd, wordlist):
    lt = dict((word, cfd[word].max()) for word in wordlist)
    baseline_tagger = nltk.UnigramTagger(model=lt,
                                         backoff=nltk.DefaultTagger('NN'))
    return baseline_tagger.evaluate(brown.tagged_sents(categories='news'))

def display():
    import pylab
    words_by_freq = list(nltk.FreqDist(brown.words(categories='news')))
    cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
    sizes = 2 ** pylab.arange(15)
    perfs = [performance(cfd, words_by_freq[:size]) for size in sizes]
    pylab.plot(sizes, perfs, '-bo')
    pylab.title('Lookup Tagger Performance with Varying Model Size')
    pylab.xlabel('Model Size')
    pylab.ylabel('Performance')
    pylab.show()

display()
```

N-gram标注器

挑选在给定上下文中最可能的标记

上下文：当前词和它前面的n-1个标识符的词性标记

问题：当标注的数据在训练数据中不存在且占比较高时，被称为数据稀疏问题，在NLP中是普遍的。因此在研究结果的精度和覆盖范围之间需要有一个权衡

```
bigram_tagger = nltk.BigramTagger(train_sents)  # 二元标注（未出现的不会统计）
bigram_tagger.tag(brown_sents[2007])

unseen_sent = brown_sents[4203]
bigram_tagger.tag(unseen_sent)

bigram_tagger.evaluate(test_sents)
```

组合标注器

如首先使用二元标注器，否则使用一元标注器，最后使用默认标注器

```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)
t2.evaluate(test_sents)

# 将会丢弃那些只看到一次或两次的上下文
t2 = nltk.BigramTagger(train_sents, cutoff=2, backoff=t1)
t2.evaluate(test_sents)
```

存储标注器

```
from pickle import dump
from pickle import load

# 存储
output = open('t2.pkl', 'wb')
dump(t2, output, -1)
output.close()

# 取出
inputs = open('t2.pkl', 'rb')
tagger = load(inputs)
inputs.close()

# 使用
text = '''
One notable effort in increasing the interoperability of biomedical
ontologies has been the creation of logical definitions[71]. This is an
initiative
'''
tokens = text.split()
tagger.tag(tokens)
```

> 给定当前单词及其前两个标记，根据训练数据，在5%的情况下，有一个以上的标记可能合理的分配给当前词。假设我们总是挑选在这种含糊不清的上下文中最可能的标记，可以得出trigram标注器性能的下界
>
> 训练数据中的歧义导致标注器性能的上限
>
> 另一种查看标注错误的方法：查看混淆矩阵

基于转换的标注 - Brill标注

思想：猜每个词的标记，然后返回和修复错误。这种方式下，Brill标注器陆续将一个不良标注的文本转换成一个更好的。同样是有监督的学习方法，不过和n-gram不同的是，它不计数观察结果，只编制一个转换修正规则的链表。

以画画类比：以大笔画开始，然后修复细节，一点点的修改

我们将研究两个规则的运作：（a）当前面的词是 TO 时，替换 NN 为 VB；（b）当下一个标记是 NNS 时，替换 TO 为 IN。表 5-6 说明了这一过程，首先使用 unigram 标注器标注，然后运用规则修正错误。

表 5-6. Brill 标注的步骤

Phrase	to	increase	grants	to	states	for	vocational	rehabilitation
Unigram	TO	NN	NNS	TO	NNS	IN	JJ	NN
Rule1		VB						
Rule2				IN				
Output	TO	VB	NNS	IN	NNS	IN	JJ	NN
Gold	TO	VB	NNS	IN	NNS	IN	JJ	NN

确定词性

> 根据词的形式（如-ness、-ing等）、根据句法（词的顺序）以及语义等综合分析。 没有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

六、学习分类文本

性别鉴定

```
import nltk
import random
from nltk.corpus import names

# 提取特征
def gender_features(word):
    return {'last_letter': word[-1]}

name_set = ([(name, 'male') for name in names.words('male.txt')] +
            [(name, 'female') for name in names.words('female.txt')])
random.shuffle(name_set)

featuresets = [(gender_features(n), g) for (n, g) in name_set]
train_set, test_set = featuresets[500:], featuresets[:500]

classifier = nltk.NaiveBayesClassifier.train(train_set)

# 查看模型效果
classifier.classify(gender_features('Neo'))
classifier.classify(gender_features('Trinity'))

nltk.classify.accuracy(classifier, test_set)

# 显示的比率为似然比，用于比较不同特征-结果关系
classifier.show_most_informative_features(5)

# 在处理大型预料库时，构建一个包含每一个实例的特征的单独的链表会使用大量的内存。
# 下述方式不会在内容中存储所有的特征集对象
from nltk.classify import apply_features
train_set = apply_features(gender_features, name_set[500:])
test_set = apply_features(gender_features, name_set[:500])
```

词性分析

```python
from nltk.corpus import brown

# 找出最常见的后缀
suffix_fdist = nltk.FreqDist()
for word in brown.words():
    word = word.lower()
    suffix_fdist[word[-1:]] += 1
    suffix_fdist[word[-2:]] += 1
    suffix_fdist[word[-3:]] += 1

common_suffixes = list(suffix_fdist.keys())[:100]
common_suffixes

# 词性特征提取器
def pos_features(word):
    features = {}
    for suffix in common_suffixes:
        features['endswith(%s)' % suffix] = word.lower().endswith(suffix)
    return features

tagged_words = brown.tagged_words(categories='news')
featuresets = [(pos_features(n), g) for (n, g) in tagged_words]

size = int(len(featuresets) * 0.1)
train_set, test_set = featuresets[size:], featuresets[:size]

classifier = nltk.DecisionTreeClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)

classifier.classify(pos_features('cats'))
```

上下文语境

```python
# 基于句子的词特征提取
# 输入分别为：句链表，句中词的索引
# 输出为特征
def pos_features(sentence, i):
    features = {'suffix(1)': sentence[i][-1:],
                'suffix(2)': sentence[i][-2:],
                'suffix(3)': sentence[i][-3:]}
    if i == 0:
        features['prev-word'] = '<START>'
    else:
        features['prev-word'] = sentence[i - 1]
    return features

brown.sents()[0]
pos_features(brown.sents()[0], 8)

tagged_sents = brown.tagged_sents(categories='news')
featuresets = []
for tagged_sent in tagged_sents:
    untagged_sent = nltk.tag.untag(tagged_sent)
    for i, (word, tag) in enumerate(tagged_sent):
        featuresets.append((pos_features(untagged_sent, i), tag))

size = int(len(featuresets) * 0.1)
```

```
train_set, test_set = featuresets[size:], featuresets[:size]

classifier = nltk.NaiveBayesClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

序列分类

```
def pos_features(sentence, i, history):
    features = {'suffix(1)': sentence[i][-1:],
                'suffix(2)': sentence[i][-2:],
                'suffix(3)': sentence[i][-3:]}
    if i == 0:
        features['prev-word'] = '<START>'
        features['prev_tag'] = '<START>'
    else:
        features['prev-word'] = sentence[i - 1]
        features['prev_tag'] = history[i - 1]
    return features

# 序列分类器
class ConsecutivePosTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = pos_features(untagged_sent, i, history)
                train_set.append((featureset, tag))
                history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = pos_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)

tagged_sents = brown.tagged_sents(categories='news')

size = int(len(tagged_sents) * 0.1)
train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]

tagger = ConsecutivePosTagger(train_sents)
tagger.evaluate(test_sents)
```

> **隐马尔可夫模型**：是统计模型，它用来描述一个含有隐含未知参数的马尔可夫过程。其难点是从可观察的参数中确定该过程的隐含参数。然后利用这些参数来作进一步的分析，例如模式识别。
>
> 在正常的马尔可夫模型中，状态对于观察者来说是直接可见的。这样状态的转换概率便是全部的参数。而在隐马尔可夫模型中,状态并不是直接可见的，但受状态影响的某些变量则是可见的。每一个状态在可能输出的符号上都有一概率分布。因此输出符号的序列能够透露出状态序列的一些信息。

> **实例**：假设你有一个住得很远的朋友，他每天跟你打电话告诉你他那天做了什么。你的朋友仅仅对三种活动感兴趣：公园散步，购物以及清理房间。他选择做什么事情只凭天气。你对于他所住的地方的天气情况并不了解，但是你知道总的趋势。在他告诉你每天所做的事情基础上，你想要猜测他所在地的天气情况。
>
> 你认为天气的运行就像一个马尔可夫链.其有两个状态 "雨"和"晴",但是你无法直接观察它们,也就是说,它们对于你是隐藏的。每天，你的朋友有一定的概率进行下列活动:"散步"、"购物"、"清理"。因为你朋友告诉你他的活动，所以这些活动就是你的观察数据。这整个系统就是一个隐马尔可夫模型HMM。
>
> 在这个例子中，如果今天下雨,那么明天天晴的概率只有30%，表示了你朋友每天做某件事的概率。如果下雨，有 50% 的概率他在清理房间；如果天晴，则有60%的概率他在外头散步。
>
> **应用**：语音识别、中文分词、光学字符识别和机器翻译等

句子分割

```
sents = nltk.corpus.treebank_raw.sents()
tokens = []  # 存储句子
boundaries = set()  # 对应句子的词索引
offset = 0  # 总词数
for sent in nltk.corpus.treebank_raw.sents():
    tokens.extend(sent)
    offset += len(sent)
    boundaries.add(offset - 1)

def punct_features(tokens, i):
    return {'next-word-capitalized': tokens[i + 1][0].isupper(),
            'prevword': tokens[i - 1].lower(),
            'punct': tokens[i],
            'prev-word-is-one-char': len(tokens[i - 1]) == 1}

# 提取可能是句子结束符的特征
# 特征 ：  句索引链表
featuresets = [(punct_features(tokens, i), (i in boundaries))
               for i in range(1, len(tokens) - 1)
               if tokens[i] in '.?!']

size = int(len(featuresets) * 0.1)
train_set, test_set = featuresets[size:], featuresets[:size]

classifier = nltk.NaiveBayesClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

七-十、文法

使用文法

```
import nltk
from nltk import CFG

groucho_grammar = CFG.fromstring("""
S -> NP VP
PP -> P NP
NP -> Det N | Det N PP | 'I'
```

```
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")

sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
# 使用递归下降分析器
parser = nltk.ChartParser(groucho_grammar)
trees = parser.parse(sent)
for tree in trees:
    print(tree)
```

```
# cfg文件内容与上块5-12行代码类似
grammar = nltk.data.load('file:mygrammar.cfg')

sent = 'Mary saw Bob'.split()
rd_parser = nltk.RecursiveDescentParser(grammar)
for tree in rd_parser.parse(sent):
    print(tree)
```

```
# 使用移进-规约分析器
sr_parse = nltk.ShiftReduceParser(grammar)
for tree in sr_parse.parse(sent):
    print(tree)

sr_parse = nltk.ShiftReduceParser(grammar, trace=2)
for tree in sr_parse.parse(sent):
    print(tree)
```

RecursiveDescentParser 递归下降分析器：

- 左递归产生式，如 NP -> NP PP ，会进行死循环
- 浪费了很多时间处理不符合输入句子的词和结构
- 回溯过程中可能会丢弃分析过的成分，它们将需要在之后再次重建

ShiftReduceParser 移进-规约分析器：

- 不执行任何回溯，所以不能保证一定能找到一个文本的解析（即使真的存在）
- 即使有多个解析最多也只能找到一个
- 每个结构只建立一次

通过优先执行规约操作来解决移进-规约冲突

交互式文法编辑器

`nltk.app.chartparser()`

依存文法

```
groucho_dep_grammar = nltk.grammar.DependencyGrammar.fromstring("""
'shot' -> 'I' | 'elephant' | 'in'
'elephant' -> 'an' | 'in'
'in' -> 'pajamas'
'pajamas' -> 'my'
```

```
""")

print(groucho_dep_grammar)

pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
sent = 'I shot an elephant in my pajamas'.split()
trees = pdp.parse(sent)
for tree in trees:
    print(tree)
```

特征结构

```
print(nltk.FeatStruct('''
[A = 'a',
B = (1)[C = 'c'],
D -> (1),
E -> (1)]
'''))

fs1 = nltk.FeatStruct(NUMBER = 74,
                      STAREET = 'run Pascal')
fs2 = nltk.FeatStruct(CITY = 'Paris')

print(fs1.unify(fs2))

fs0 = nltk.FeatStruct(A = 'a')
fs1 = nltk.FeatStruct(A = 'b')
fs2 = fs0.unify(fs1)
print(fs2)

fs1 = nltk.FeatStruct('''
[ADDRESS1 = [NUMBER = 74, STREET = 'run Pascal']]
''')
fs2 = nltk.FeatStruct('''
[ADDRESS1 = ?x,
ADDRESS2 = ?x]
''')
print(fs1)
print(fs2)
print(fs2.unify(fs1))
```

十一、语言数据管理

```
s1 = '01010101'
s2 = '00001111'
s3 = '00001110'

# 在指定窗口大小下计算两个字符串的差异大小
nltk.windowdiff(s1, s2, 3)
nltk.windowdiff(s2, s3, 3)
```

TIMIT

> 语音语料库，数据是文本形式

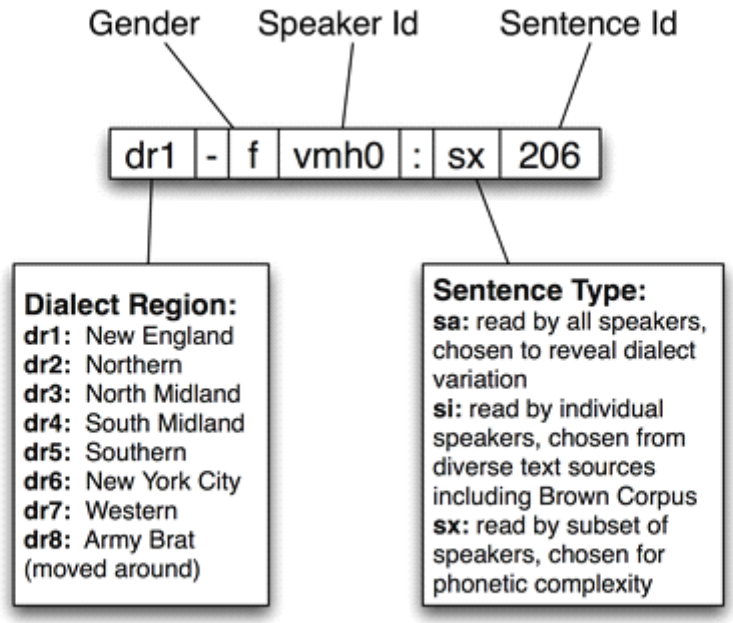

图 11-1. TIMIT 标识符的结构：每个记录使用说话者的方言区、性别、说话者标识符、句子类型、句子标识符组成的一个字符串作为标签。

```
import nltk

phonetic = nltk.corpus.timit.phones('dr1-fvmh0/sa1')
nltk.corpus.timit.word_times('dr1-fvmh0/sa1')

timitdict = nltk.corpus.timit.transcription_dict()
timitdict['greasy']

nltk.corpus.timit.spkrinfo('dr1-fvmh0')
```

XML

```
merchant_file = nltk.data.find('corpora/shakespeare/merchant.xml')
raw = open(merchant_file).read()
print(raw[0: 168])

from xml.etree import ElementTree

merchant = ElementTree.parse(merchant_file)
merchant

speaker_seq = [s.text for s in
merchant.findall('ACT/SCENE/SPEECH/SPEAKER')]
speaker_freq = nltk.FreqDist(speaker_seq)
top5 = list(speaker_freq.keys())[:5]
top5
```

Toolbox

```
from nltk.corpus import toolbox

lexicon = toolbox.xml('rotokas.dic')
# 1、索引访问
lexicon[3][0]
lexicon[3][0].tag
```

```
lexicon[3][0].text

# 2、路径访问
[lexeme.text.lower() for lexeme in lexicon.findall('record/lx')][:10]

# 格式化
html = '<table>\n'
for entry in lexicon[70: 80]:
    lx = entry.findtext('lx')
    ps = entry.findtext('ps')
    ge = entry.findtext('ge')
    html += '  <tr><td>%s</td><td>%s</td><td>%s</td></tr>\n' % (lx, ps, ge)

html += '</table>'
print(html)

# 为每个条目计算字段的平均个数
from nltk.corpus import toolbox

lexicon = toolbox.xml('rotokas.dic')
sum(len(entry) for entry in lexicon) / len(lexicon)
```

OLAC元数据

元数据：关于数据的结构化数据

OLAC：开放语言档案社区