**NLP\_chapter7**

**第七章 从文本提取信息**

**7.1 信息提取**

信息有很多种“形状”和“大小”。一个重要的形式是**结构化数据**:实体和关系的可预测的规范的结构。例如一个表格。

从文本中获得相似的信息比较麻烦，解决方法之一是建立一个非常通用的意义重现(第 10 章)。在这一章中，采取不同的方法，提前定为将只查找文本中非常具体的各种信息，如:组织和地点之间的关系。即首先将自然语言句子这样的**非结构化数据**转换成类似表格的结构化数据。然后，利用强大的查询工具，如 SQL。这种从文本获取意义的方法被称为**信息提取**。

**信息提取结构**

信息提取系统的简单的流水线结构：

首先，使用句子分割器将该文档的原始文本分割成句，使用分词器将每个句子进一步细分为词。接下来，对每个句子进行词性标注，在下一步**命名实体识别**中将证明这是非常有益的。在这一步，寻找每个句子中提到的潜在的有趣的实体。最后，使用**关系识别**搜索文本中不同实体间的可能关系。生成(entity, relation, entity)元组的一个链表作为输出。

要执行前面三项任务，可以定义一个函数，简单地连接 NLTK 中默认的句子分割器，分词器和词性标注器:

def ie\_preprocess(document):

sentences = nltk.sent\_tokenize(document)

sentences = [nltk.word\_tokenize(sent) for sent in sentences]

sentences = [nltk.pos\_tag(sent) for sent in sentences]

接下来，命名实体识别中，分割和标注可能组成一个有趣关系的实体。通常情况下，这些将被定义为名词短语，例如 the knights who say "ni"或者适当的名称如 Monty Python。在一些任务中，同时考虑不明确的名词或名词块也是有用的，如 every student 或 cats，这些不必要一定与定义 NP 和适当名称一样的方式指示实体。

最后，在提取关系时，搜索对文本中出现在附近的**实体对**之间的特殊模式，并使用这些模式建立元组记录实体之间的关系。

**7.2 分块**

用于实体识别的基本技术是**分块(chunking)**，分割和标注多标识符序列。小框显示词级标识符和词性标注，大框显示较高级别的分块。每个这种较大的框叫做**一大块(chunk)**。就像分词忽略空白符，分块通常选择标识符的一个子集。同样像分词一样，分块构成的源文本中的片段不能重叠。

**名词短语分块**

名词短语分块，或 NP-分块(NP-chunking)，即找单独名词短语对应的块。

NP-块往往是比完整的名词短语小的片段。这种差异的动机之一是 NP-块被定义为不包含其他的 NP-块。因此，修饰一个名词的任何介词短语或从句将不包括在相应的 NP-块内，因为它们几乎可以肯定包含更多的名词短语。

NP-分块信息最有用的来源之一是词性标记。这是在信息提取系统中进行词性标注的动机之一。为了创建一个 NP-块，首先定义一个**块语法**，规定句子应如何分块。在下例中，将用一个正则表达式规则定义一个简单的语法。这条规则是说一个 NP-块由一个可选的限定词(DT) 后面跟着任何数目的形容词(JJ)然后是一个名词(NN)组成。使用此语法，创建了一个块分析器，测试例句。结果是一棵树，可以输出或图形显示：

sentence = [("the","DT"),("little","JJ"),("yellow","JJ"),

("dog","NN"),("barked","VBD"),("at","IN"),("the","DT"),("cat","NN")]

grammar = "NP:{<DT>?<JJ>\*<NN>}"

cp = nltk.RegexpParser(grammar)

result = cp.parse(sentence)

print(result)

result.draw()

**标记模式**

组成一个块语法的规则使用**标记模式**来描述已标注的词的序列。一个标记模式是一个用尖括号分隔的词性标记序列，如<DT>?<JJ>\*<NN>。标记模式类似于正则表达式模式。

更复杂一点的标记模式，如<DT>?<JJ.\*>\*<NN.\*>+。这将把任何以一个可选的限定词开头，后面跟零个或多个任何类型的形容词(包括相对形容词，如 earlier/JJR)，后面跟一个或多个任何类型的名词的标识符序列分块。然而依旧存在许多该规则不包括的更复杂的例子，需要进一步完善，可以使用图形界面 nltk.app.chunkpa rser()进行测试。

**用正则表达式分块**

要找到一个给定的句子的块结构，RegexpParser 分块器以一个没有标识符被分块的平面结构开始，轮流应用分块规则，依次更新块结构。所有的规则都被调用后，返回块结构。

下例显示了一个由 2 个规则组成的简单的块语法。第一条规则匹配一个可选的限定词或所有格代名词，零个或多个形容词，然后跟一个名词；第二条规则匹配一个或多个专有名词：

grammar = r'''

NP: {<DT|PP\$>?<JJ>\*<NN>}

{<NNP>+}

'''

cp = nltk.RegexpParser(grammar)

sentence = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"),

("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]

print(cp.parse(sentence))

注意：$符号是正则表达式中的一个特殊字符，必须使用转义符\来匹配PP$标记。

如果标记模式匹配位置重叠，最左边的匹配优先。例如:如果应用一个匹配两个连续的名词文本的规则到一个包含三个连续的名词的文本，则只有前两个名词将分块:

nouns = [("money","NN"),("market","NN"),("fund","NN")]

grammar = "NP:{<NN><NN>}"

cp = nltk.RegexpParser(grammar)

print(cp.parse(nouns))

当为块规则添加注释时，分块器将它作为其跟踪输出的一部分输出这些注释。

**探索文本语料库**

可以使用分块器更容易地在已标注的语料库中提取匹配的特定的词性标记序列的短语:

def find\_chunks(chunk):

cp = nltk.RegexpParser(chunk)

for sent in brown.tagged\_sents()[:20]:

tree = cp.parse(sent)

for subtree in tree.subtrees():

if subtree.label() == 'CHUNK':

print(subtree)

elif subtree.label() == 'NOUNS':

print(subtree)

find\_chunks('CHUNK: {<V.\*><TO><V.\*>}')

find\_chunks('NOUNS: {<N.\*>{4,}}')

**加缝隙**

有时定义想从一个块排除什么比较容易。可以为不包括在一大块中的一个标识符序列定义一个**缝隙**。在下面的例子中，barked/VBD at/IN是一个缝隙:

[ the/DT little/JJ yellow/JJ dog/NN ] barked/VBD at/IN [ the/DT cat/NN ]

加缝隙是从一大块中去除一个标识符序列的过程。如果匹配的标识符序列贯穿一整块，那么这一整块会被去除;如果标识符序列出现在块中间，这些标识符会被去除，在以前只有一个块的地方留下两个块。如果序列在块的周边，这些标记被去除，留下一个较小的块。

简单的加缝器:

grammar = r'''

NP:

{<.\*>+}

}<VBD|IN>+{

'''

sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),

("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]

cp = nltk.RegexpParser(grammar)

print(cp.parse(sentence))

**块的表示:标记与树**

作为标注和分析之间的中间状态，块结构可以使用标记或树来表示。使用最广泛的表示是 **IOB 标记**。在这个方案中，每个标识符被用三个特殊的块标签之一标注，I(inside，内部)，O(outside，外部)或B(begin ，开始)。一个标识符被标注为B，如果它标志着一个块的开始。块内的标识符子序列被标注为I。所有其他的标识符被标注为O。B和 I标记加块类型的后缀，如 B-NP, I-NP。当然，没有必要指定出现在块外的标识符的类型，所以这些都只标注为O。

IOB 标记已成为文件中表示块结构的标准方式。

在文件中表示时，每个标识符一行，和它的词性标记与块标记一起。这种格式允许表示多个块类型，只要块不重叠。正如前面所看到的，块的结构也可以使用树表示。这有利于使每块作为一个组成部分可以直接操作。

NLTK 中树作为块的内部表示，并提供这些树与 IOB 格式互换的方法。

**7.3 开发和评估分块器**

**读取 IOB 格式与 CoNLL2000 分块语料库**

使用 corpus 模块，可以加载已标注的《华尔街日报》文本，然后使用 IOB 符号分块。这个语料库提供的块类型有 NP，VP 和 PP，每个句子使用多行表示。

转换函数 chunk.conllstr2tree()用这些多行字符串建立一个树表示。此外，它允许使用三个块类型的任何子集：

nltk.chunk.conllstr2tree(text,chunk\_types=['NP']).draw()

如果分块后的文本中包含函数中没有加入的块类型，那么树图将不显示他的块类型，但会以跟O型相同的形式出现在图中。

可以使用 NLTK的 corpus 模块访问较大量的已分块文本。CoNLL2000 分块语料库包含 27 万词的《华尔街日报文本》，分为“训练”和“测试”两部分，标注有词性标记和IOB 格式分块标记。可以使用 nltk.corpus.conll2000 访问这些数据。下面是一个读取语料库的“训练”部分的 100 个句子的例子:

from nltk.corpus import conll2000

print(conll2000.chunked\_sents('train.txt')[99])

CoNLL2000 分块语料库包含三种块类型:NP 块;VP块;PP 块。因为现在唯一感兴趣的是 NP 块，可以使用 chunk\_types 参数选择它们:

print(conll2000.chunked\_sents('train.txt',chunk\_types=['NP'])[99])

**简单评估和基准**

为琐碎的不创建任何块的块分析器 cp 建立一个基准(baseline):

cp = nltk.RegexpParser('')

test\_sents = conll2000.chunked\_sents('test.txt',chunk\_types=['NP'])

print(cp.evaluate(test\_sents))

IOB 标记准确性表明超过三分之一的词被标注为 O，即没有在 NP 块中。然而，由于标注器没有找到任何块，其精度、召回率和 F-度量均为零。尝试一个初级的正则表达式分块器，查找以名词短语标记的特征字母(如 CD、DT 和 JJ)开头的标记:

grammar = r"NP:{<[CDJNP].\*>+}"

cp = nltk.RegexpParser(grammar)

print(cp.evaluate(test\_sents))

采用更多数据驱动的方法改善它，使用训练语料找到对每个词性标记最有可能的块标记(I、O 或 B)。换句话说，可以使用 unigram 标注器建立一个分块器。但不是尝试确定每个词的正确的词性标记，而是给定每个词的词性标记，尝试确定正确的块标记。

在下例中，定义了 UnigramChunker 类，使用 unigram 标注器给句子加块标记。这个类的大部分代码只是用来在 NLTK的 ChunkParserI接口使用的分块树表示和嵌入式标注器使用的 IOB 表示之间镜像转换。类定义了两个方法:一个构造函数，当我们建立一个新的 UnigramChunker 时调用;一个 parse 方法，用来给新句子分块：

class UnigramChunker(nltk.ChunkParserI):

def \_\_init\_\_(self,train\_sents):

train\_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(sent)]

for sent in train\_sents]

self.tagger = nltk.UnigramTagger(train\_data)

def parse(self,sentence):

pos\_tags = [pos for (word,pos) in sentence]

tagged\_pos\_tags = self.tagger.tag(pos\_tags)

chunktags = [chunktag for (pos,chunktag) in tagged\_pos\_tags]

conlltags = [(word,pos,chunktag) for ((word,pos),chunktag)

in zip(sentence,chunktags)]

return nltk.chunk.conlltags2tree(conlltags)

构造函数需要训练句子的一个链表，这将是块树的形式。它首先将训练数据转换成适合训练标注器的形式，使用 tree2conlltags 映射每个块树到一个词，标记，块三元组的链表。然后使用转换好的训练数据（仅包括标记和块）训练一个 unigram 标注器，并存储在 self.tagger供以后使用。

parse 方法取一个已标注的句子作为其输入，以从那句话提取词性标记开始。然后使用在构造函数中训练过的标注器 self.tagger，为词性标记标注 IOB 块标记。接下来，提取块标记，与原句组合，产生 conlltags。最后，使用 conlltags2tree 将结果转换成一个块树。

现在使用 CoNLL2000 分块语料库训练UnigramChunker，并测试其性能:

test\_sents = conll2000.chunked\_sents('test.txt',chunk\_types=['NP'])

train\_sents = conll2000.chunked\_sents('train.txt',chunk\_types=['NP'])

unigram\_chunker = UnigramChunker(train\_sents)

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

看一看通过使用 unigram 标注器分配一个标记给每个语料库中出现的词性标记，它学到了什么:

postags = sorted(set(pos for sent in train\_sents

for (word,pos) in sent.leaves()))

print(unigram\_chunker.tagger.tag(postags))

它已经发现大多数标点符号出现在 NP 块外，除了两种货币符号#和$。它也发现限定词(DT)和所有格(PRP$和 WP$)出现在 NP 块的开头，而名词类型(NN, NNP, NNPS,NNS)大多出现在 NP 的块内。

同理，建立一个 bigram 分块器，只需要改变类的名称为 BigramChunker，修改原例中行：

self.tagger = nltk.UnigramTagger(train\_data)

构造一个 BigramTagger而不是 UnigramTagger。由此产生的分块器的性能略高于 unigram 分块器:

bigram\_chunker = BigramChunker(train\_sents)

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

**训练基于分类器的分块器**

无论是基于正则表达式的分块器还是 n-gram 分块器，决定创建什么块完全基于词性标记。然而，有时词性标记不足以确定一个句子应如何分块。

如果想最大限度地提升分块的性能，需要使用词的内容信息作为词性标记的补充。

包含词的内容信息的方法之一是使用基于分类器的标注器对句子分块。

基于分类器的 NP 分块器的基础代码如下例所示。它包括两个类:第一个类几乎与上一章第五节例子中 ConsecutivePosTagger类相同。仅有的两个区别是它调用一个不同的特征提取 器，使用 MaxentClassifier 而不是 NaiveBayesClassifier(这里因为未能成功安装megam文件，最大熵分类器无法运行，所以仍使用朴素贝叶斯分类器代替);第二个类基本上是标注器类的一个包装器，将它变成一个分块器。训练期间，这第二个类映射训练语料中的块树到标记序列;在 parse()方法中，它将标注器提供的标记序列转换回一个块树：

class ConsecutiveNPChunkTagger(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 = npchunk\_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 = npchunk\_features(sentence,i,history)

tag = self.classifier.classify(featureset)

history.append(tag)

return zip(sentence,history)

class ConsecutiveNPChunker(nltk.ChunkParserI):

def \_\_init\_\_(self,train\_sents):

tagged\_sents = [[((w,t),c) for (w,t,c) in

nltk.chunk.tree2conlltags(sent)]

for sent in train\_sents]

self.tagger = ConsecutiveNPChunkTagger(tagged\_sents)

def parse(self,sentence):

tagged\_sents = self.tagger.tag(sentence)

conlltags = [(w,t,c) for ((w,t),c) in tagged\_sents]

return nltk.chunk.conlltags2tree(conlltags)

接下来唯一需要填写的是特征提取器。首先，定义一个简单的特征提取器，它只是提供了当前标识符的词性标记。使用此特征提取器，基于分类器的分块器的性能与 unigram 分块器非常类似:

def npchunk\_features(sentence,i,history):

word,pos = sentence[i]

return {"pos": pos}

chunker = ConsecutiveNPChunker(train\_sents)

print(chunker.evaluate(test\_sents))

还可以添加一个特征:前面词的词性标记。添加此特征允许分类器模拟相邻标记之间的相互作用，由此产生的分块器与 bigram 分块器非常接近：

def npchunk\_features(sentence,i,history):

word,pos = sentence[i]

if i == 0:

prevword,prevpos = "<START>","<START>"

else:

prevword,prevpos = sentence[i-1]

return {"pos":pos,"prevpos":prevpos}

chunker = ConsecutiveNPChunker(train\_sents)

print(chunker.evaluate(test\_sents))

下一步，尝试把当前词增加为特征，因为假设这个词的内容应该对分块有用。这个特征确实提高了分块器的性能，大约 1.5 个百分点(相应的错误率减少大约 10%):

def npchunk\_features(sentence,i,history):

word,pos = sentence[i]

if i == 0:

prevword,prevpos = "<START>","<START>"

else:

prevword,prevpos = sentence[i-1]

return {"pos":pos,"word":word,"prevpos":prevpos}

chunker = ConsecutiveNPChunker(train\_sents)

print(chunker.evaluate(test\_sents))

最后，尝试用多种附加特征扩展特征提取器，例如:预取特征、配对功能和复杂的语境特征。这最后一个特征，被称为 tags-since-dt，创建一个字符串，描述自最近的限定词以来遇到的所有词性标记:

def npchunk\_features(sentence,i,history):

word,pos = sentence[i]

if i == 0:

prevword,prevpos = "<START>","<START>"

else:

prevword,prevpos = sentence[i-1]

if i == len(sentence)-1:

nextword,nextpos = "<END>","<END>"

else:

nextword,nextpos = sentence[i+1]

return {"pos":pos,

"word":word,

"prevpos":prevpos,

"nextpos":nextpos,

"prevpos+pos":"{}+{}".format(prevpos,pos),

"pos+nextpos":"{}+{}".format(pos,nextpos),

"tags-since-dt":tags\_since\_dt(sentence,i)}

def tags\_since\_dt(sentence,i):

tags = set()

for word,pos in sentence[:i]:

if pos == 'DT':

tags = set()

else:

tags.add(pos)

return '+'.join(sorted(tags))

chunker = ConsecutiveNPChunker(train\_sents)

print(chunker.evaluate(test\_sents))

还可以继续为特征提取器函数 npchunk\_features 增加不同的特征，进一步改善NP 分块器的性能。

**7.4 语言结构中的递归**

**用级联分块器构建嵌套结构**

到目前为止，块结构一直是相对平的。已标注标识符组成的树在如 NP 这样的块节点下任意组合。然而，只需创建一个包含递归规则的多级的块语法，就可以建立任意深度的块结构。下例是名词短语、介词短语、动词短语和句子的模式。这是一个四级块语法器，可以用来创建深度最多为 4 的结构：

grammar = r'''

NP:{<DT|JJ|NN.\*>+}

PP:{<IN><NP>}

VP:{<VB.\*><NP|PP|CLAUSE>+$}

CLAUSE:{<NP><VP>}

'''

cp = nltk.RegexpParser(grammar)

sentence = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),

("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

print(cp.parse(sentence))

这一语法存有一些缺陷，它无法识别包含从句的从句或其他块。

这些问题的解决方案是:让分块器在它的模式中循环:尝试完所有模式之后，重复此过程。添加一个可选的第二个参数 loop 指定这套模式应该循环的次数:

cp = nltk.RegexpParser(grammar,loop=2)

print(cp.parse(sentence))

这个级联过程使我们能创建深层结构。然而，创建和调试级联过程是困难的，关键点是它能更有效地做全面的分析(见第 8 章)。另外，级联过程只能产生固定深度的树(不超过级联级数)，对于完整的句法分析来说这是不够的。

**树**

树是一组连接的加标签节点，从一个特殊的**根节点**沿一条唯一的路径到达**每个节点**。注意:它们标准的画法是颠倒的，即从每个节点到根节点，但在UI界面上往往相反。

用“家庭”来比喻树中节点的关系:例如:S 是 VP 的**父母**;反之 VP 是 S 的一个**孩子**。此外，由于 NP 和 VP 同为 S 的两个孩子，它们也是**兄弟**。也有特定树的文本格式，正如之前所有print语句所打印出来的。

虽然我们将只集中关注语法树，但是树可以用来编码任何同构的超越语言形式序列的层次结构(如形态结构、篇章结构)。一般情况下，叶子和节点值不一定要是字符串。

在 NLTK 中创建一棵树，通过给一个节点添加标签和一个孩子链表:

tree1 = nltk.Tree('NP',['Alice'])

print(tree1)

tree2 = nltk.Tree('NP',['the','rabbit'])

print(tree2)

可以将这些不断合并成更大的树:

tree3 = nltk.Tree('VP',['chased',tree2])

tree4 = nltk.Tree('S',[tree1,tree3])

print(tree4)

注意，不同于之前自动生成的树，这里打印出的树没有进行换行和缩进。

树对象的一些的方法:

print(tree4[1])

print(tree4[1].label())

print(tree4.leaves())

print(tree4[1][1][1])

复杂的树用括号表示难以阅读。在这些情况下，draw 方法是非常有用的。它会打开一个新窗口，包含树的一个图形表示。树显示窗口可以放大和缩小，子树可以**折叠和展开**，并将图形表示输出为一个 postscript 文件(包含在一个文档中)。

tree4.draw()

**树遍历**

使用递归函数来遍历树是标准的做法:

def traverse(t):

try:

t.label()

except AttributeError:

print(t,end=' ')

else:

print('(',t.label(),end=' ')

for child in t:

traverse(child)

print(')',end=' ')

t = '(S (NP Alice) (VP chased (NP the rabbit)))'

traverse(t)

在这里使用了一种叫做动态类型的技术，检测 t 是一棵树(如:定义了 t.node )。

**7.5 命名实体识别**

命名实体是确切的名词短语，指示特定类型的个体，如组织、人、日期等。下表列出了一些较常用的 NEs 类型。这些应该是不言自明的，除了“FACILITY”:建筑和土木工程领域的人造产品;以及“GPE”:地缘政治实体，如城市、州/省、国家。

表 常用命名实体类型

NE类型 例子

组织 Georgia-Pacific Corp., WHO

人 Eddy Bonte, President Obama

地点 Murray River , Mount Everest

日期 June, 2008-06-29

时间 two fifty a m, 1:30 p.m.

货币 175 million Canadian Dollars, GBP 10.40 百分数 twenty pct, 18.75 %

设施 Washington Monument, Stonehenge 地缘政治实体 South East Asia, Midlothian

**命名实体识别**(NER)系统的目标是识别所有文字提及的命名实体。可以分解成两个子任务:确定 NE 的边界和确定其类型。命名实体识别经常是信息提取中关系识别的前奏，它也有助于其他任务。例如:在问答系统(QA)中，试图提高信息检索的精确度，不是返回整个页面而只是包含用户问题的答案的那些部分。大多数 QA 系统利用标准信息检索返回文件，然后尝试分离文档中包含答案的最小的文本片段。

识别命名实体的一个办法是查找一个适当的名称列表。例如:识别地点时，可以使用地名辞典，如亚历山大地名辞典或盖蒂地名辞典。然而，盲目这样做会出问题，比如辞典中的某些词本身含有其他含义，在文本的处理中就会出错。

人或组织的名称的情况更加困难。任何这些名称的列表都肯定覆盖不全。每天都有新的组织出现，如果是处理当代文本或博客条目，使用名称辞典查找来识别众多实体是不可能的。

困难的另一个原因是许多命名实体措辞有歧义。

更大的挑战来自如 Stanford University 这样的多词名称和包含其他名称的名称。因此，在命名实体识别中，需要能够识别多标识符序列的开头和结尾。

命名实体识别是一个非常适合用基于分类器类型的方法来处理的任务，这些方法在名词短语分块时出现过。特别是，可以建立一个标注器，为使用 IOB 格式的每个块都加了适当类型标签的句子中的每个词加标签。

在CONLL 2002(conll2002)荷兰语训练数据中，每个标识符一行，与它的词性标记及命名实体标记一起。基于这个训练语料，可以构造一个可以用来标注新句子的标注器，使用 nltk.chunk.conlltags2tree()函数将标记序列转换成一个块树。

NLTK提供了一个已经训练好的可以识别命名实体的分类器，使用函数 nltk.ne\_chunk()访问。如果设置参数 binary=True，那么命名实体只被标注为NE;否则，分类器会添加类型标签，如 PERSON, ORGANIZATION, and GPE：

sent = nltk.corpus.treebank.tagged\_sents()[22]

print(nltk.ne\_chunk(sent,binary=True))

print(nltk.ne\_chunk(sent))

**7.6 关系抽取**

一旦文本中的命名实体已被识别，就可以提取它们之间存在的关系。如前所述，通常会寻找**指定类型的**命名实体之间的关系。进行这一任务的方法之一是首先寻找所有 (X, α, Y)形式的三元组，其中 X 和 Y 是指定类型的命名实体，α表示 X 和 Y 之间关系的字符串。然后使用正则表达式从α的实体中抽出正在查找的关系。下面的例子搜索包含词 in 的字符串。特殊的正则表达式(?!\b.+ing\b)是一个否定预测先行断言，允许忽略如 success in supervising the transition of 中的字符串，其中 in 后面跟一个动名词:

IN = re.compile(r'.\*\bin\b(?!\b.+ing)')

for doc in nltk.corpus.ieer.parsed\_docs('NYT\_19980315'):

for rel in nltk.sem.extract\_rels('ORG','LOC',doc,

corpus='ieer',pattern=IN):

print(nltk.sem.rtuple(rel))

搜索关键字 in 执行的相当不错，虽然它的检索结果也会误报，例如: [ORG: House Transportation Committee] , secured the most money in the [LOC: New York];一种简单的基于字符串的方法排除这样的填充字符串似乎不太可能。

如前文所示，CoNLL 2002 命名实体语料库的荷兰语部分不只包含命名实体标注，也包含词性标注。这允许我们设计对这些标记敏感的模式，如下面的例子所示。show\_clause()方法以分条形式输出关系，其中二元关系符号作为参数 relsym 的值被指定：

from nltk.corpus import conll2002

vnv = '''

(

is/V|

was/V|

werd/V|

wordt/V

)

.\*

van/Prep

'''

VAN = re.compile(vnv,re.VERBOSE)

for doc in conll2002.chunked\_sents('ned.train'):

for r in nltk.sem.extract\_rels('PER','ORG',doc,

corpus='conll2002',pattern=VAN):

print(nltk.sem.clause(r,relsym='VAN'))

print(nltk.sem.rtuple(r,lcon=True,rcon=True))