# 第3章 词性标注

在上一章中，我们对自己所要做的所有预处理步骤进行了讨论，以便在工作中可以应对任何文本语料库。我们现在应该可以放心地对任何种类的文本进行解析和清理了。我们应该执行所有的文本预处理，譬如针对任意文本的断词处理、词干提取以及停用词移除等。我们可以根据自己的需要执行和定制所有相关的预处理工具。到目前为止，我们已经重点讨论了针对文本型文档的一般性预处理工作。现在，让我们将焦点转向那些动作更为激烈的NLP预处理步骤吧。

在本章，我们将具体讨论何谓词性标注，以及词性（POS）在NLP应用环境中的意义。我们也会学习如何用NLTK标注有意义的信息，并介绍可用于NLP密集型应用程序的各种标注器。最后，我们还将学习如何用NLTK来标注命名实体。我们会详细讨论各种NLP标注器，并且还会提供一些代码片段来帮助您理解它们。我们也将会看到这些标注器的最佳实践，以说明在什么地方应该使用哪种标注器。在读完本章之后，读者应了解：

* 何谓词性标注，以及其在NLP中的重要性。
* 如何使用NLTK中形形色色的词性标注。
* 如何用NLTK创建自定义的词性标注。

## 何谓词性标注

其实，我们可能在小时候就已经听说过**词性（POS）**这个术语了。尤其在形容词和副词的实际使用上，要想掌握其中的窍门还是很花时间的。这两者的区别究竟是什么？或许，我们可以考虑将所有这方面的知识进行编码以创建一个系统。这件事看起来好像挺容易的的，但这几十年来，将这些知识转化为可编码的机器学习模型一直都是一个非常难解的NLP问题。在我个人看来，虽然目前最先进的词性标注算法在预测给定单词的词性上已经有了较高的精确度（约97％）。但词性标注领域中仍有大量的研究在等着我们。

对于像英语这样的语言来说，它们在新闻和其他领域往往都有许多已被标注的语料库。这为我们带来了许多先进的算法。尽管在一般情况下，这其中的一些标注器应该足以应付各种跨不同领域的、文本化的使用环境了。但在某些特定的用例中，POS的预判可能还是有些不尽如人意。对于这些用例，我们可能就得要从头开始建立一个标注器了。然而，如果我们想要深入了解POS，就得先要对机器学习领域中的一些技术有一个基本的了解。虽然这部分有些是我们要在*第6章：文本分类*中要讨论的内容，但在这里，我们必须讨论一下相关基础知识，以便我们可以创建一个自定义的POS标注器。

首先，我们要学习一些现成可用的POS标注器，及其相配的token集。在此，我们会看到一些以**元组**形式存在的、独立单词的POS。然后，我们再将焦点转移到其中一些标注其的内部工作原理上来，最后，我们还将讨论如何从头开始创建一个自定义的标注器。

我们在讨论POS时，总少不了会用到Penn Treebank[[1]](#footnote-1)这个最常用到的POS标记库：

|  |  |
| --- | --- |
| **标签** | **相关说明** |
| NNP | 专用名词的单数形式 |
| NNPS | 专用名词的复数形式 |
| PDT | 前置限定词 |
| POS | 所有格结束符 |
| PRP | 人称代词 |
| PRP$ | 所有格代词 |
| RB | 副词 |
| RBR | 相对副词 |
| RBS | 最高级副词 |
| RP | 小品词 |
| SYM | 符号（数学符号或特殊符号） |
| TO | To |
| UH | 叹词 |
| VB | 动词的基本形式 |
| VBD | 动词的过去式 |
| VBG | 动词的动名词用法 |
| VBN | 动词的过去分词 |
| WP | Wh-代词 |
| WP$ | 所有格wh-代词 |
| WRB | Wh-副词 |
| # | 井号符 |
| $ | 美元符 |
| . | 句号 |
| , | 逗号 |
| : | 分号，分隔符 |
| ( | 左括号 |
| ) | 右括号 |
| " | 直双引号 |
| ' | 左单引号 |
| " | 左双引号 |
| ' | 右单引号 |
| " | 右双引号 |

这些看起来就是我们在小学英语课堂上所学的东西，对吧？现在，既然我们已经了解了这些标签所代表的含义，下面就可以来运行一个实验了：

>>> import nltk

>>> from nltk import word\_tokenize

>>> s = "I was watching TV"

>>> print nltk.pos\_tag(word\_tokenize(s))

[('I', 'PRP'), ('was', 'VBD'), ('watching', 'VBG'), ('TV', 'NN')]

如果我们只想使用新闻类或与其类似语料库的POS，那么您就只需要知道前三行代码即可。在这段代码中，我们先对一段文本进行了断词处理，然后对其调用了NLTK库中的pos\_tag方法，得到了一组（词形，词性标签）形式的元组。这就是一个NLTK库内置的POS标注器。

|  |
| --- |
| 该方法在内部使用的是由maxent分类器（我们会在后面章节中讨论分类器这个话题）所调校出来的模型，以预测特定单词属性所属于的类型标签。  如果您想了解更多其中的细节，可以参考下面这个链接： https://github.com/nltk/nltk/blob/develop/nltk/ tag/\_\_init\_\_.py |

由于NLTK库使用了Python中强大而有效的数据结构，所以我们在使用由NLTK库处理之后所输出的成果时会具有更多的灵活性。现在，您一定很想见识一下POS在真实应用中的典型用法究竟会是怎样的。在一个典型的预处理操作中，我们通常要尽可能地找出所有的名词。下面我们来看一段代码，它会帮我们找出给定句子中的所有名词：

>>> tagged = nltk.pos\_tag(word\_tokenize(s))

>>> allnoun = [word for word,pos in tagged if pos in ['NN','NNP'] ]

现在，我们来试着回答一下下面的问题：

* 我们可以在进行词性标注之前先删除掉停用词吗？
* 我们要如何获取到该句子中所有的动词？

### Stanford标注器

NLTK库还有一个非常棒的的特性，就是它还有许多针对其它的预置标注器的封装器，例如**Stanford工具包**。下面的POSTagger就一个很常见的例子：

>>> from nltk.tag.stanford import POSTagger

>>> import nltk

>>> stan\_tagger = POSTagger('models/english-bidirectional-distdim. tagger','standford-postagger.jar')

>>> tokens = nltk.word\_tokenize(s)

>>> stan\_tagger.tag(tokens)

|  |
| --- |
| 如果您想要使用上述代码，就必须先从下面的链接中获取Stanford标注器： http://nlp.stanford.edu/software/stanford-postagger-full-2014-08-27.zip.  然后将其中的jar文件和模型文件解压到一个文件夹中，然后在POSTagger()的参数中指定其绝对路径。 |

现在我们来总结一下，用NLTK库实现标注任务的方式主要有两种：

1. 使用NLTK库或其他库中的预置标注器，并将其运用到测试数据上。这两种标注器应该足以应付纯英语文本环境，以及非特殊领域语料库中所有的词性标注任务了。
2. 基于测试数据来创建或调校出适用的标注器。这就等于我们要处理一个非常特殊的用例或者开发一个自定义的标注器了。

下面，我们来更深入地了解一下一个典型的POS标注器在其内部究竟做了些什么。

### 深入了解标注器

一个典型的标注器通常要用到大量的调校数据，它主要被用于标注出句子中的各个单词，并为其标上POS标签。标注是个纯手动的操作，看起来应该像这样：

Well/UH what/WP do/VBP you/PRP think/VB about/IN the/DT idea/NN of/IN ,/, uh/UH ,/, kids/NNS having/VBG to/TO do/VB public/JJ service/NN work/NN for/IN a/DT year/NN ?/.Do/VBP you/PRP think/VBP it/PRP 's/BES a/DT ,/,

上面这些样例取自Penn Treebank总机的语料库。人们已经针对一些大型语料库的做了大量的手动标注工作。甚至我们还有一个叫**做语言数据联盟**（**简称LDC**）[[2]](#footnote-2)组织，那里的人们就花了很多时间来研究不同语言的标注、不同的文本种类以及不同的标注操作，譬如词性标注，句法分析标注、以及对话标注等（我们会在后面讨论这些话题）。

|  |
| --- |
| 读者可以从<https://www.ldc.upenn.edu/>处获得上述所有的资源以及更多的相关信息。（尽管LDC可以免费提供一小部分数据的一小部分，但如果需要我们也可以购买去整个标注语料库。NLTK库中大概内置了PTB的大约10％的数据。） |

如果我们还希望调校出属于自己的POS标注器，就需要针对特定的领域来执行相关的标注操作。这种标注操作就需要有这些领域的专家来协助。

通常情况下，像词性标注这样的标注问题往往会被视同为顺序性标签问题或者某种分类问题，后者特指人们为特定token所生成的正确标签，并用相关判别模型对其进行预判的那一类问题。

为了不让读者直接掉入过于复杂的应用实例中，我们先从一些简答的标注方法开始。

下面我们来看一段代码，它将会告诉我们Brown语料库中各POS标签的分布频率：

>>> from nltk.corpus import brown

>>> import nltk

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

>>> print nltk.FreqDist(tags)

< FreqDist: 'NN': 13162, 'IN': 10616, 'AT': 8893, 'NP': 6866, ',': 5133,

'NNS': 5066, '.': 4452, 'JJ': 4392 >

如您所见，NN是这里出现频繁最高的标签，我们可以基于此来创建一个非常幼稚的POS标注器，用于给所有的测试文本分配NN标签。 当然，我们也可以使用NLTK中的一个叫做DefaultTagger的函数来做这件事。DefaultTagger函数就是个顺序性标注器，下面我们就来讨论一下这个标注器。这里会调用到一个evaluate()函数，主要用于评估相关单词POS的准确度。这是我们针对Brown语料库的标注器所用的基准。在default\_tagger这个案例中，我们的预测正确率大约在13％左右。下面，我们将会把同样的基准进一步推广到所有的标注器上。

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

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

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

0.130894842572

### 顺序性标注器

毫无疑问，上面那个标识器简直是弱爆了。其实，DefaultTagger本质上只是基类SequentialBackoffTagger的一部分而已，后者是一个顺序性的标注服务。标注器会试着基于其所处的上下文环境来模型化相关的标签。而且如果它不能进行正确的标签预测，就会去咨询BackoffTagger。通常情况下，DefaultTagger参数都可以被当作一个BackoffTagger实体来使用。

下面，让我们继续来看一些更复杂的顺序性标注器。

#### N-gram标注器

**N-gram** tagger is a subclass of SequentialTagger, where the tagger takes previous *n* words in the context, to predict the POS tag for the given token. There are variations of these taggers where people have tried it with UnigramsTagger, BigramsTagger, and TrigramTagger:

的N-gram标注器是SequentialTagger，其中所述标记器需要先前n个字中的背景下，以预测在POS标签对于给定令牌的子类。有这些标注器，人们曾与UnigramsTagger，BigramsTagger和TrigramTagger尝试过的变化：

>>> from nltk.tag import UnigramTagger

>>> from nltk.tag import DefaultTagger

>>> from nltk.tag import BigramTagger

>>> from nltk.tag import TrigramTagger # we are dividing the data into a test and train to evaluate our taggers.

>>> train\_data = brown\_tagged\_sents[:int(len(brown\_tagged\_sents) \* 0.9)]

>>> test\_data = brown\_tagged\_sents[int(len(brown\_tagged\_sents) \* 0.9):]

>>> unigram\_tagger = UnigramTagger(train\_data,backoff=default\_tagger)

>>> print unigram\_tagger.evaluate(test\_data)

0.826195866853

>>> bigram\_tagger = BigramTagger(train\_data, backoff=unigram\_tagger)

>>> print bigram\_tagger.evaluate(test\_data)

0.835300351655

>>> trigram\_tagger = TrigramTagger(train\_data,backoff=bigram\_tagger)

>>> print trigram\_tagger.evaluate(test\_data)

0.83327713281

Unigram just considers the conditional frequency of tags and predicts the most frequent tag for the every given token. The bigram\_tagger parameter will consider the tags of the given word and the previous word, and tag as tuple to get the given tag for the test word. The TrigramTagger parameter looks for the previous two words with a similar process.

单字只考虑标签的条件频率和预测为每一个给定的令牌最常见的标签。该bigram\_tagger参数将考虑给定的字和前一个词，而标签的标签，展示的元组来获取测试字给定的标签。该TrigramTagger参数查找前两个单词，一个类似的过程。

It's very evident that coverage of the TrigramTagger parameter will be less and the accuracy of that instance will be high. On the other hand, UnigramTagger will have better coverage. To deal with this tradeoff between precision/recall, we combine the three taggers in the preceding snippet. First it will look for the trigram of the given word sequence for predicting the tag; if not found it Backoff to BigramTagger parameter and to a UnigramTagger parameter and in end to a NN tag.

这是非常明显的是，TrigramTagger参数的覆盖范围会越来越该实例的精度将是高的。另一方面，UnigramTagger将有更好的覆盖。为了应对精度/召回之间的这种权衡，我们在前面的代码片段三者有机结合起来标注器。首先，它会寻找预测变量的给定单词序列的卦;如果没有发现退避到BigramTagger参数和一个UnigramTagger参数，并在结束一个NN标签。

#### 正则表达式标注器

There is one more class of sequential tagger that is a regular expression based taggers. Here, instead of looking for the exact word, we can define a regular expression, and at the same time we can define the corresponding tag for the given expressions. For example, in the following code we have provided some of the most common regex patterns to get the different parts of speech. We know some of the patterns related to each POS category, for example we know the articles in English and we know that anything that ends with *ness* will be an adjective. Instead, we will write a bunch of regex and a pure python code, and the NLTK RegexpTagger parameter will provide an elegant way of building a pattern based POS. This can also be used to induce domain related POS patterns.

还有一个班连续恶搞的是一个基于正则表达式标注器。在这里，而不是寻找确切的词，我们可以定义一个正则表达式，并在同一时间，我们可以定义表达式给出相应的标记。例如，在下面的代码我们已经提供了一些最常见的正则表达式模式来获得语音的不同部分。我们知道一些与每个POS类的模式，比如我们知道的英文文章，我们知道，任何以内斯结束将是一个形容词。相反，我们将写一堆正则表达式和纯Python代码，而NLTK RegexpTagger参数将提供构建基于模式的POS一种优雅的方式。这也可以用于诱导域相关的POS图案。

>>> from nltk.tag.sequential import RegexpTagger

>>> regexp\_tagger = RegexpTagger(

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

( r'(The|the|A|a|An|an)$', 'AT'), # articles

( r'.\*able$', 'JJ'), # adjectives

( r'.\*ness$', 'NN'), # nouns formed from adj

( r'.\*ly$', 'RB'), # adverbs

( r'.\*s$', 'NNS'), # plural nouns

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

(r'.\*ed$', 'VBD'), # past tense verbs

(r'.\*', 'NN') # nouns (default)

])

>>> print regexp\_tagger.evaluate(test\_data)

0.303627342358

We can see that by just using some of the obvious patterns for POS we are able to reach approximately 30 percent in terms of accuracy. If we combine regex taggers, such as the BackoffTagger, we might improve the performance. The other use case for regex tagger is in the preprocessing step, where instead of using a raw Python function string.sub(), we can use this tagger to tag date patterns, money patterns, location patterns and so on.

我们可以看到，通过只使用一些用于POS的明显图案的，我们能够在精度方面达到约30％。如果我们结合正则表达式标注器，如BackoffTagger，我们可能会提高性能。其他使用案例的正则表达式标注器是在预处理步骤，它不使用原始的Python函数string.sub（），我们可以利用这个恶搞标记日期模式，资金模式，位置图案等。

* Can you modify the code of a hybrid tagger in the N-gram tagger section to work with Regex tagger? Does that improve performance?
* Can you write a tagger that tags Date and Money expressions?
* 你可以修改的混合恶搞的代码在N-克恶搞部分与正则表达式恶搞的工作？这是否提高性能？
* 你可以写标签日期和货币表达式恶搞？

### Brill标注器

Brill tagger is a transformation based tagger, where the idea is to start with a guess for the given tag and, in next iteration, go back and fix the errors based on the next set of rules the tagger learned. It's also a supervised way of tagging, but unlike N-gram tagging where we count the N-gram patterns in training data, we look for transformation rules.

布里尔恶搞是基于变换恶搞，这里的想法是开始为指定代码的猜测，并在接下来的迭代，回去收拾基于下一组规则的恶搞学到的错误。这也是标注的监督方式，但不同的N-gram标签，我们在训练数据计数的N-gram模式，我们期待为转换规则。

If the tagger starts with a Unigram / Bigram tagger with an acceptable accuracy, then brill tagger, instead looking for a trigram tuple, will be looking for rules based on tags, position and the word itself.

如果恶搞与一单字/两字组恶搞具有可接受的精度开始，然后布里尔恶搞，而不是寻找一个卦元组，将寻找基于标签，位置和这个词本身的规则。

An example rule could be:

一个例子规则可以是：

Replace NN with VB when the previous word is TO.

当上一个字是用VB替换NN。

After we already have some tags based on UnigramTagger, we can refine if with just one simple rule. This is an interactive process. With a few iterations and some more optimized rules, the brill tagger can outperform some of the N-gram taggers. The only piece of advice is to look out for over-fitting of the tagger for the training set.

之后，我们已经有了基于UnigramTagger一些标签，我们可以用，如果只是一个简单的规则细化。这是一个互动的过程。随着几次迭代以及一些更优化的规则，布里尔恶搞可以超越一些的N-gram标注器的。建议的唯一的一块是看出来的过度拟合打标签的训练集。

|  |
| --- |
| You can also look at the work here for more example rules.  您还可以看看这里的工作更多的例子规则。  <http://stp.lingfil.uu.se/~bea/publ/megyesiBrillsPoSTagger.pdf> |

* Can you try to write more rules based on your observation?
* Try to combine brill tagger with UnigramTagger.
* 你可以尝试编写基于你的观察更多的规则？
* 尝试布里尔恶搞与单字标注器结合起来。

### 基于标注器的机器学习

Until now we have just used some of the pre-trained taggers from NLTK or Stanford. While we have used them in the examples in previous section, the internals of the taggers are still a black box to us. For example, pos\_tag internally uses a **Maximum Entropy Classifier** (**MEC**). While StanfordTagger also uses a modified version of Maximum Entropy. These are discriminatory models. While there are many **Hidden Markov Model** (**HMM**) and **Conditional Random Field** (**CRF**) based taggers, these are generative models.

到现在为止，我们刚刚使用了一些来自NLTK或斯坦福的预先训练标注器的。尽管我们在上一节的示例中使用它们，则标注器的内部仍然是一个黑盒子给我们。例如，pos\_tag内部使用最大熵分类（MEC）。虽然StanfordTagger还采用最大熵的修改版本。这是歧视性的车型。虽然有许多隐马尔可夫模型（HMM）和条件随机场（CRF）的标注器，这些都是生成模型。

Covering all of these topics is beyond the scope of the book. I would highly recommend the NLP class for a great understanding of these concepts. We will cover some of the classification techniques in *Chapter 6*, *Text Classification*, but some of these are very advanced topics in NLP, and will need more attention.

涵盖了所有的这些话题已经超出了本书的范围。我会极力推荐的NLP类的这些概念非常了解。我们将介绍一些在第6章，文本分类的分类技术，但其中也不乏一些非常高级的主题在NLP，并且需要更多的关注。

If I have to explain in short, the way to categorize POS tagging problem is either as a classification problem where given a word and the features like previous word, context, morphological variation, and so on. We classify the given word into a POS category, while the others try to model it as a generative model using the similar features. It's for the reader's reference to go over some of these topics using links in the tips.

如果非要在短期解释，顺便分类词性标注的问题是无论是作为在那里给一个单词，像前一个单词，语境，形态变异等特点分类问题。我们给定的字分类成POS类别，而其他尝试将其建模为使用类似功能的生成模型。这对读者参考介绍一些使用中的提示链接这些主题。

|  |
| --- |
| NLP CLASS: https://www.coursera.org/course/nlp  HMM: http://mlg.eng.cam.ac.uk/zoubin/papers/ ijprai.pdf  MEC: <https://web.stanford.edu/class/cs124/lec/Maximum_Entropy_Classifiers.pdf>  <http://nlp.stanford.edu/software/tagger.shtml> |

## Named Entity Recognition (NER)

Aside from POS, one of the most common labeling problems is finding entities in the text. Typically NER constitutes name, location, and organizations. There are NER systems that tag more entities than just three of these. The problem can be seen as a sequence, labeling the Named entities using the context and other features. There is a lot more research going on in this area of NLP where people are trying to tag Biomedical entities, product entities in retail, and so on. Again, there are two ways of tagging the NER using NLTK. One is by using the pre-trained NER model that just scores the test data, the other is to build a Machine learning based model. NLTK provides the ne\_chunk() method and a wrapper around Stanford NER tagger for Named Entity Recognition.

除了POS，最常见的标签的问题之一是要找到在文本实体。通常情况下NER构成的名称，位置和组织。有该标签的不仅仅是这三种多个实体NER系统。这个问题可以看作是一个序列，标签使用上下文和其它特征的命名实体。还有很多更多的研究在自然语言处理这个领域，人们正试图标记生物实体，产品实体零售，等事情。同样，也有使用标记的NLTK NER两种方式。一种是通过使用预先训练NER模式，只是分数测试数据，另一个是建立一个基于机器学习模型。 NLTK提供了ne\_chunk（）方法，并围绕斯坦福NER恶搞的包装用于命名实体识别。

### NER tagger

NLTK provides a method for Named Entity Extraction: ne\_chunk. We have shown a small snippet to demonstrate how to use it for tagging any sentence. This method will require you to preprocess the text to tokenize for sentences, tokens, and POS tags in the same order to be able to tag for Named entities. NLTK used ne\_chunking, where chunking is nothing but tagging multiple tokens to a call it a meaningful entity.

除了POS，最常见的标签的问题之一是要找到在文本实体。通常情况下NER构成的名称，位置和组织。有该标签的不仅仅是这三种多个实体NER系统。这个问题可以看作是一个序列，标签使用上下文和其它特征的命名实体。还有很多更多的研究在自然语言处理这个领域，人们正试图标记生物实体，产品实体零售，等事情。同样，也有使用标记的NLTKNER两种方式。一种是通过使用预先训练NER模式，只是分数测试数据，另一个是建立一个基于机器学习模型。NLTK提供了ne\_chunk（）方法，并围绕斯坦福NER恶搞的包装用于命名实体识别。

NE chunking is loosely used in the same way as Named entity:

网元分块中相同的方式命名实体泛指：

>>> import nltk

>>> from nltk import ne\_chunk

>>> Sent = "Mark is studying at Stanford University in California"

>>> print(ne\_chunk(nltk.pos\_tag(word\_tokenize(sent)), binary=False))

(S (PERSON Mark/NNP) is/VBZ studying/VBG at/IN (ORGANIZATION Stanford/NNP University/NNP) in/IN NY(GPE California/NNP)))

The ne\_chunking method recognizes people (names), places (location), and organizations. If binary is set to True then it provides the output for the entire sentence tree and tags everything. Setting it to False will give us detailed person, location and organizations information, as with the preceding example using the Stanford NER Tagger.

该ne\_chunking方法识别人（姓名），地点（地点），和组织。如果二进制设置为True，那么它为整个句子的树和标签的一切输出。它设置为False将会给我们详细的人，地点和组织的信息，与使用斯坦福NER标注器前面的例子。

Similar to the POS tagger, NLTK also has a wrapper around Stanford NER. This NER tagger has better accuracy. The code following snippet will let you use the tagger. You can see in the given example that we are able to tag all the entities with just three lines of code:

类似于POS恶搞，NLTK还拥有斯坦福大学周围NER的包装。这NER恶搞具有更高的精度。该代码下面的代码片断将让你用恶搞。你可以在给定的例子，我们能够只用三行代码标记所有实体看看：

>>> from nltk.tag.stanford import NERTagger

>>> st = NERTagger('<PATH>/stanford-ner/classifiers/all.3class.distsim. crf.ser.gz',... '<PATH>/stanford-ner/stanford-ner.jar')

>>> st.tag('Rami Eid is studying at Stony Brook University in NY'.split())

[('Rami', 'PERSON'), ('Eid', 'PERSON'), ('is', 'O'), ('studying', 'O'),

('at', 'O'), ('Stony', 'ORGANIZATION'), ('Brook', 'ORGANIZATION'),

('University', 'ORGANIZATION'), ('in', 'O'), ('NY', 'LOCATION')]

If you observe closely, even with a very small test sentence, we can say Stanford Tagger outperformed the NLTK ne\_chunk tagger.

如果你仔细观察，即使有一个非常小的考验句话，我们可以说斯坦福标注器跑赢NLTK ne\_chunk恶搞。

Now, these kinds of NER taggers are a nice solution for a generic kind of entity tagging, but we have to train our own tagger, when it comes, to tag domain specific entities like biomedical and product names, so we have to build our own NER system. I would also recommend an NER Calais. It has ways of tagging not just typical NER, but also some more entities. The performance of this tagger is also very good:

现在，这类NER标注器是一个通用的一种实体标记的一个很好的解决方案，但我们必须培养我们自己的恶搞，当它来临的时候，标记像生物医学和产品名称特定领域的实体，所以我们要建立我们自己NER系统。我还建议在NER加莱。它有标注不只是典型的NER，但也有一些更多的实体的方法。这个恶搞的表现也很不错：

https://code.google.com/p/python-calais/

## 该让你练练手了

Here are the answers to the questions posed in the above sections:

这里的答案在上面部分所提出的问题：

* Can we remove stop words before POS tagging?
* 我们能词性标注之前删除停用词？

No; If we remove the stop words, we will lose the context, and some of the POS taggers (Pre-Trained model) use word context as features to give the POS of the given word.

没有;如果我们去掉停用词，我们将失去的背景下，以及一些POS标注器（预先训练模型）的用字方面的特点给定单词的POS。

* How can we get all the verbs in the sentence?
* 我们怎样才能在句子中的所有动词？

We can get all the verbs in the sentence by using pos\_tag

我们可以通过使用pos\_tag得到句子所有动词

>>>tagged = nltk.pos\_tag(word\_tokenize(s))

>>>allverbs = [word for word,pos in tagged if pos in

['VB','VBD','VBG'] ]

* Can you modify the code of the hybrid tagger in the N-gram tagger section to work with Regex tagger? Does that improve performance?
* 您可以修改混合恶搞的代码在N-克恶搞部分与正则表达式恶搞的工作？这是否提高性能？

Yes. We can modify the code of the hybrid tagger in the N-gram tagger section to work with the Regex tagger:

是。我们可以修改的N-gram恶搞部分与正则表达式恶搞上班混合恶搞的代码：

>>>print unigram\_tagger.evaluate(test\_data,backoff= regexp\_tagger)

>>>bigram\_tagger = BigramTagger(train\_data, backoff=unigram\_ tagger) >>>print bigram\_tagger.evaluate(test\_data)

>>>trigram\_tagger=TrigramTagger(train\_data,backoff=bigram\_tagger)

>>>print trigram\_tagger.evaluate(test\_data)

0.857122212053

0.866708415627

0.863914446746

The performance improves as we add some basic pattern-based rules, instead of predicting the most frequent tag.

* Can you write a tagger that tags Date and Money expressions? Yes, we can write a tagger that tags Date and Money expressions. Following is the code:
* 你可以写标签日期和货币表达式恶搞？是的，我们可以编写标记日期和货币表达式恶搞。以下是代码：

>>>date\_regex = RegexpTagger([(r'(\d{2})[/.-](\d{2})[/.-](\d{4})$'

,'DATE'),(r'\$','MONEY')])

>>>test\_tokens = "I will be flying on sat 10-02-2014 with around

10M $ ".split()

>>>print date\_regex.tag(test\_tokens)

|  |
| --- |
| The last two questions haven't been answered.  最后两个问题没有得到回答。  There can be many rules according to the reader's observation, so there is no Right / Wrong answer here.  可以有根据读者的意见，许多规则，所以没有正确/错误答案在这里。 |

Can you try a similar word cloud to what we did in *Chapter 1*, *Introduction to Natural Language Processing* with only nouns and verbs now?

你可以尝试类似的词云来我们在第1章，自然语言处理，只有名词和动词现在做？

References:

引用资料：

<https://github.com/japerk/nltk-trainer>

<http://en.wikipedia.org/wiki/Part-of-speech_tagging>

<http://en.wikipedia.org/wiki/Named-entity_recognition>

<http://www.inf.ed.ac.uk/teaching/courses/icl/nltk/tagging.pdf>

<http://www.nltk.org/api/nltk.tag.html>

## 本章小结

This chapter was intended to expose the reader to some of the most useful NLP pre-processing steps of tagging. We have talked about the Part of Speech problem in general, including the significance of POS in the context of NLP. We also discussed the different ways we can use a pre-trained POS tagger in NLTK, how simple it is to use, and how to create wonderful applications. We then talked about all the available POS tagging options, like N-gram tagging, Regex based tagging, etc. We have developed a mix of these taggers that can be built for domain specific corpuses.

本章是旨在向读者揭露的一些标记的最有用的NLP预处理步骤。我们一般谈到言语问题的部分，包括POS机的NLP背景下的意义。我们还讨论了不同的方式，我们可以使用一个预先训练POS恶搞在NLTK，这是​​多么简单的使用，以及如何创建精彩应用。然后，我们谈到了所有可用的词性标注的选项，如N元标记，基于正则表达式标记，等我们开发这些标注器，可以为特定领域的语料建成的混合。

We briefly talked about how a typical pre-trained tagger is built. We discussed the possible approaches to address tagging problems. We also talked about NER taggers, and how it works with NLTK. I think if, by the end of this chapter, the user understands the importance of POS and NER in general in the context of NLP, as well as how to run the snippet of codes using NLTK, I will consider this chapter successful. But the journey does not end here. We know some of the shallow NLP preprocessing steps now, and in most of the practical application POS, the NER predominantly used. In more complex NLP applications such as the Q/A system, Summarization, and Speech we need deeper NLP techniques like Chunking, Parsing, Semantics. We will talk about these in the next chapter.

我们简要地谈到了一个典型的预先训练恶搞是如何构建的。我们讨论了可能的方法来解决标记问题。我们也谈到了NER标注器，以及它如何与NLTK工作。我想，如果通过本章的最后，用户理解POS和NER的普遍在自然语言处理中的重要性，以及如何运行使用NLTK代码的片段，我会考虑这一章成功。但旅程并没有到此结束。我们现在知道一些浅NLP预处理步骤，并且在大多数的实际应用的POS的，所述NER主要使用。在更复杂的NLP应用，如Q / A系统，汇总和演讲，我们需要更深入的NLP技术，如组块，解析，语义。我们将在下一章谈论这些。

1. 译者注：Penn Treebank原本是一个NLP项目的名称，该项目主要用于对相关语料进行标注，标注内容包括词性标注以及句法分析。其语料来源是1989年的华尔街日报，包含了2499篇文章。这里指代该项目所标注的结果。 [↑](#footnote-ref-1)
2. 译者注：LDC，全名Linguistic Data Consortium，这是一个由大学、图书馆、企业、政府、研究机构共同合办的联合企业，成立于1992年，目前由宾夕法尼亚大学负责主要运营。LDC最初的角色只是保存与分发科研要用到的语言数据，后来有了资金，就开始自行收集并构建一些数据，如今该组织已经拥有了非常多的语言数据资源，成为了最主要的科研语言资源管理分发机构之一。 [↑](#footnote-ref-2)