# 第3章 词性标注

In previous chapters, we talked about all the preprocessing steps we need, in order to work with any text corpus. You should now be comfortable about parsing any kind of text and should be able to clean it. You should be able to perform all text preprocessing, such as Tokenization, Stemming, and Stop Word removal on any text. You can perform and customize all the preprocessing tools to fit your needs. So far, we have mainly discussed generic preprocessing to be done with text documents. Now let's move on to more intense NLP preprocessing steps.

在前面的章节中，我们谈到了我们需要的所有预处理步骤，为了与任何语料库工作。您现在应该是舒适有关分析任何类型的文本，应该能够进行清洁。您应该能够执行所有文本预处理，如任何文字符号化，词干和停止词删除。您可以执行和自定义所有的预处理工具，以满足您的需求。到目前为止，我们已经重点讨论通用预处理与文本文档来完成。现在，让我们进入更激烈的NLP预处理步骤。

In this chapter, we will discuss what part of speech tagging is, and what the significance of POS is in the context of NLP applications. We will also learn how to use NLTK to extract meaningful information using tagging and various taggers used for NLP intense applications. Lastly, we will learn how NLTK can be used to tag a named entity. We will discuss in detail the various NLP taggers and also give a small snippet to help you get going. We will also see the best practices, and where to use what kind of tagger. By the end of this chapter, readers will learn:

在本章中，我们将讨论什么词性标注的部分是，什么POS的意义是在NLP应用的上下文。我们还将学习如何使用NLTK提取使用标签和用于NLP密集型应用程序的各种标注器有意义的信息。最后，我们将学习如何NLTK可以用来标记命名实体。我们将详细讨论各种NLP标注器，也给一个小片段来帮助你走了。我们也将看到最好的做法，何地使用何种恶搞的。通过本章的最后，读者将了解到：

* What is Part of speech tagging and how important it is in context of NLP
* What are the different ways of doing POS tagging using NLTK
* How to build a custom POS tagger using NLTK
* 什么是词性标注以及如何重要的是它在自然语言处理的情况下
* 有什么做用NLTK词性标注方式的不同
* 如何建立使用NLTK定制POS恶搞

## 何谓词性标注

In your childhood, you may have heard the term **Part of Speech** (**POS**). It can really take good amount of time to get the hang of what adjectives and adverbs actually are. What exactly is the difference? Think about building a system where we can encode all this knowledge. It may look very easy, but for many decades, coding this knowledge into a machine learning model was a very hard NLP problem. I think current state of the art POS tagging algorithms can predict the POS of the given word with a higher degree of precision (that is approximately 97 percent). But still lots of research going on in the area of POS tagging.

在你的童年，你可能已经听说语音（POS）的任期一部分。它可以真正需要的时间量好得到一个什么样的形容词和副词其实都是挂起。究竟有什么区别？想想建立一个系统，我们可以编码所有这方面的知识。它可能看起来很容易，但几十年来，编码这些知识转化为机器学习模型是一个非常困难的NLP问题。我认为艺术词性标注算法目前的状态可以预测具有较高的精确度的定单词的POS（即约97％）。但是还是很多的研究词性标注的面积怎么回事。

Languages like English have many tagged corpuses available in the news and other domains. This has resulted in many state of the art algorithms. Some of these taggers are generic enough to be used across different domains and varieties of text. But in specific use cases, the POS might not perform as expected. For these use cases, we might need to build a POS tagger from scratch. To understand the internals of a POS, we need to have a basic understanding of some of the machine learning techniques. We will talk about some of these in *Chapter 6*, *Text Classification*, but we have to discuss the basics in order to build a custom POS tagger to fit our needs.

英语等语言在新闻和其他领域提供许多标记语料库。这导致了本领域的算法许多状态。一些这些标注器是足够通用在不同的域和品种文本的情况下使用。但在具体的使用情况下，可能POS未如预期完成。对于这些使用情况下，我们可能需要从头开始建立一个POS恶搞。要了解一个POS机的内部，我们需要有一些的机器学习技术有基本的了解。我们将讨论其中的一些在第6章，文本分类，但我们必须讨论，以建立一个自定义的POS恶搞，以适应我们的需要的基础知识。

First, we will learn some of the pertained POS taggers available, along with a set of tokens. You can get the POS of individual words as a **tuple**. We will then move on to the internal workings of some of these taggers, and we will also talk about building a custom tagger from scratch.

首先，我们将学习一些可用的pertained POS标注器，用一组令牌一起。你可以得到单个的单词作为一个元组的POS机。然后，我们将移动到的一些标注器的内部工作，我们也将谈论从头开始建立一个自定义的恶搞。

When we talk about POS, the most frequent POS notification used is Penn Treebank:

当我们谈论POS，是最常用的POS通知宾州树库：

|  |  |
| --- | --- |
| **Tag** | **Description** |
| NNP | Proper noun, singular |
| NNPS | Proper noun, plural |
| PDT | Pre determiner |
| POS | Possessive ending |
| PRP | Personal pronoun |
| PRP$ | Possessive pronoun |
| RB | Adverb |
| RBR | Adverb, comparative |
| RBS | Adverb, superlative |
| RP | Particle |
| SYM | Symbol (mathematical or scientific) |
| TO | To |
| UH | Interjection |
| VB | Verb, base form |
| VBD | Verb, past tense |
| **Tag** | **Description** |
| VBG | Verb, gerund/present participle |
| VBN | Verb, past |
| WP | Wh-pronoun |
| WP$ | Possessive wh-pronoun |
| WRB | Wh-adverb |
| # | Pound sign |
| $ | Dollar sign |
| . | Sentence-final punctuation |
| , | Comma |
| : | Colon, semi-colon |
| ( | Left bracket character |
| ) | Right bracket character |
| " | Straight double quote |
| ' | Left open single quote |
| " | Left open double quote |
| ' | Right close single quote |
| " | Right open double quote |

Looks pretty much like what we learned in primary school English class, right? Now once we have an understanding about what these tags mean, we can run an experiment:

看起来很像我们在小学英语课堂上所学的吧？现在，一旦我们有什么这些标签的意思理解，我们可以运行一个实验：

>>>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')]

If you just want to use POS for a corpus like news or something similar, you just need to know the preceding three lines of code. In this code, we are tokenizing a piece of text and using NLTK's pos\_tag method to get a tuple of (word, pos-tag). This is one of the pre-trained POS taggers that comes with NLTK.

如果你只是想用POS像新闻或类似的东西语料库，你只需要知道前面三行代码。在这段代码中，我们的记号化一块文本，并使用NLTK的pos\_tag方法来获取（词形，词性标记）的元组。这是一个带有NLTK的预先训练POS标注器之一。

|  |
| --- |
| It's internally using the maxent classifier (will discuss these classifiers in advanced chapters) trained model to predict to which class of tag a particular word belongs.  它的内部使用了MAXENT分类培训的（将讨论高级章节这些分类）模型预测到标签类特定的单词所属。  To get more details you can use the following link:  为了让你可以使用下面的链接的详细信息：  https://github.com/nltk/nltk/blob/develop/nltk/ tag/\_\_init\_\_.py |

NLTK has used python's powerful data stru ctures efficiently, so we have a lot more flexibility in terms of use of the results of NLTK outputs. You must be wondering what could be a typical use of POS in a real application. In a typical preprocessing, we might want to look for all the nouns. Now this code snippet will give us all the nouns in the given sentence:

NLTK已经有效地使用Python的强大的数据STRU ctures，所以我们在使用NLTK产出的成果方面有更多的灵活性。你一定想知道什么可能是在实际应用中的典型使用POS机。在一个典型的预处理，我们可能要寻找所有的名词。现在，这个代码片段将使我们在给定的句子中的所有名词：

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

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

Try to answer the following questions:

试着回答以下问题：

* Can we remove stop words before POS tagging?
* How can we get all the verbs in the sentence?
* 我们能词性标注之前删除停用词？
* 我们怎样才能在句子中的所有动词？

### Stanford标注器

Another awesome feature of NLTK is that it also has many wrappers around other pre-trained taggers, such as **Stanford tools**. A common example of a POS tagger is shown here:

NLTK的另一个真棒特点是，它也有各地的预先训练标注器，如斯坦福大学的工具很多包装。一个POS恶搞的一个常见的例子如下所示：

>>>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)

|  |
| --- |
| To use the above code, you need to download the Stanford tagger from  使用上面的代码，你需要从下载斯坦福恶搞  http://nlp.stanford.edu/software/stanford-postagger-full-2014-08-27.zip.  Extract both the jar and model into a folder, and give an absolute path in argument for the POSTagger.  同时抽取的jar和模型到一个文件夹，并给予论证的POS标注器的绝对路径。 |

Summarizing this, there are mainly two ways to achieve any tagging task in NLTK:

这个总结，主要有两种方式来实现任何标记任务NLTK：

1. Using NLTK's or another lib's pre-trained tagger, and applying it on the test data. Both preceding taggers should be sufficient to deal with any POS tagging task that deals with plain English text, and the corpus is not very domain specific.
2. Building or Training a tagger to be used on test data. This is to deal with a very specific use case and to develop a customized tagger.
3. 使用NLTK的或其他的lib的预先训练标注器，并应用其上的测试数据。前两者应该标注器足以应付任何词性标注任务与纯英文文本交易，胼不是很特定领域。
4. 建立或培训捉人要对测试数据使用。这是处理一个非常具体的用例和开发定制的恶搞。

Let's dig deeper into what goes on inside a typical POS tagger.

让我们深入挖掘典型的POS恶搞里面发生的事情。

### 深入了解标注器

A typical tagger uses a lot of trained data, with sentences tagged for each word that will be the POS tag attached to it. Tagging is purely manual and looks like this:

一个典型的恶搞采用了大量训练有素的数据，用标记每个单词，这将是它相连的POS标记的句子。标记是纯手工的，看起来像这样：

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 ,/,

The preceding sample is taken from the Penn Treebank switchboard corpus. People have done lot of manual work tagging large corpuses. There is a **Linguistic Data Consortium (LDC)** where people have dedicated so much time to tagging for different languages, different kinds of text and different kinds of tagging like POS, dependency parsing, and discourse (will talk about these later).

前面的样品从宾州树库语料总机拍摄。人们已经做了很多事情标注大型语料库的。有一个地方的人有专门的这么多的时间来标记不同的语言，不同的文字和不同种类的标签像POS，依存分析，和话语（会谈论这些更高版本）的语言数据联盟（LDC）。

|  |
| --- |
| You can get all these resources and more information about them at https://www.ldc.upenn.edu/. (LDC provides a fraction of data for free but you can also purchase the entire tagged corpus. NLTK has approximately 10 percent of the PTB.)  你可以得到所有这些资源以及关于它们在https://www.ldc.upenn.edu/更多信息。 （LDC免费提供数据的一小部分，但你也可以购买整个标注语料。NLTK有PTB的大约10％。） |

If we also want to train our own POS tagger, we have to do the tagging exercise for our specific domain. This kind of tagging will require domain experts.

如果我们也想培养我们自己的POS恶搞，我们要做的标记锻炼我们的特定领域。这种标签将需要领域的专家。

Typically, tagging problems like POS tagging are seen as sequence labeling problems or a classification problem where people have tried generative and discriminative models to predict the right tag for the given token.

通常情况下，像词性标注标注问题被视为序列标注问题，或者人们试图生成和判别模型来预测正确的标签为给定令牌的分类问题。

Instead of jumping directly in to more sophisticated examples, let's start with some simple approaches for tagging.

而不是直接跳跃到更复杂的例子，让我们开始与标注一些简单的方法。

The following snippet gives us the frequency distribution of POS tags in the Brown corpus:

下面的代码片段让我们在布朗语料库POS标签的频率分布：

>>>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 >

We can see NN comes as the most frequent tag, so let's start building a very naïve POS tagger, by assigning NN as a tag to all the test words. NLTK has a DefaultTagger function that can be used for this. DefaultTagger function is part of the Sequence tagger, which will be discussed next. There is a function called evaluate() that gives the accuracy of the correctly predicted POS of the words. This is used to benchmark the tagger against the brown corpus. In the default\_tagger case, we are getting approximately 13 percent of the predictions correct. We will use the same benchmark for all the taggers moving forward.

我们可以看到NN当属最频繁的标签，让我们开始建立一个非常幼稚的POS恶搞，通过分配NN作为标记的所有测试的话。 NLTK具有可用于该一个DefaultTagger功能。 DefaultTagger功能是顺序标记器，这将在下面讨论的一部分。有一个称为功能评价（），让这些词的正确预测的POS的准确性。这是用来基准对褐语料库打标签。在default\_tagger情况下，我们正在正确的预测约13％。我们将使用相同的基准所有标注器前进。

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

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

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

0.130894842572

### 顺序标注器

Not surprisingly, the above tagger performed poorly. The DefaultTagger is part of a base class SequentialBackoffTagger that serves tags based on the Sequence. Tagger tries to model the tags based on the context, and if it is not able to predict the tag correctly, it consults a BackoffTagger. Typically, the DefaultTagger parameter could be used as a BackoffTagger.

毫无疑问，上述恶搞表现不佳。该DefaultTagger是一个基类SequentialBackoffTagger供应基础上，序列标签的一部分。捉人者试图根据上下文的标签模型，并且如果它不能够正确地预测该标记，将查询一个BackoffTagger。典型地，DefaultTagger参数可以作为一个BackoffTagger。

Let's move on to more sophisticated sequential taggers.

让我们移动到更复杂的顺序标注器。

#### 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。

### Machine learning based tagger

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.

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.

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.

|  |
| --- |
| 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.

### 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.

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

>>>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.

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:

>>>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.

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:

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

## Your Turn

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

* Can we remove stop words before POS tagging?

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.

* How can we get all the verbs in the sentence?

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

>>>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?

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

>>>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:

>>>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.  There can be many rules according to the reader's observation, so there is no Right / Wrong answer here. |

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?

References:

<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>

## Summary

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.

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.