# 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.

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:

* 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

## What is Part of speech tagging

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.

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.

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.

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

|  |  |
| --- | --- |
| **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:

**>>>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.

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.

To get more details you can use the following link:

[https://github.com/nltk/nltk/blob/develop/nltk/ tag/\_\_init\_\_.py](https://github.com/nltk/nltk/blob/develop/nltk/tag/__init__.py)

NLTK has used python's powerful data structures 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:

**>>>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:

* Can we remove stop words before POS tagging?
* How can we get all the verbs in the sentence?

### Stanford tagger

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:

**>>>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

[http://nlp.stanford.edu/software/stanford-postagger-](http://nlp.stanford.edu/software/stanford-postagger-full-2014-08-27.zip)

[full-2014-08-27.zip](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.

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

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

### Diving deep into a tagger

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:

**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).

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.)

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.

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.

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

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

**>>>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.

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

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

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

**0.130894842572**

### Sequential tagger

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.

Let's move on to more sophisticated sequential taggers.

#### N-gram tagger

**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:

**>>>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.

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.

#### Regex tagger

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.

**>>>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.

* 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?

### Brill tagger

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.

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.

An example rule could be:

Replace NN with VB when the previous word is TO.

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.

You can also look at the work here for more example rules.

[http://stp.lingfil.uu.se/~bea/publ/megyesiBrillsPoSTagger.pdf](http://stp.lingfil.uu.se/~bea/publ/megyesi-BrillsPoSTagger.pdf)

* Can you try to write more rules based on your observation?
* Try to combine brill tagger with UnigramTagger.

### 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](http://mlg.eng.cam.ac.uk/zoubin/papers/ijprai.pdf)

MEC: [https://web.stanford.edu/class/cs124/lec/ Maximum\_Entropy\_Classifiers.pdf](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.