Getting Started with Natural Language Toolkit (NLTK)

An interactive tutorial for beginners

(UIUC MCSDS CS410 Technology Review Project by John Moran and Graham Chester)

Authors

John Moran - jfmoran2: Division of Labor, 50% share: jointly reviewed existing tutorials for NLTK, Stopwords and Tokenizers: jointly worked on text and coding, Stemming and Lemmatization: text and coding, Parts of Speech Tagging: text and coding.

Graham Chester - grahamc2: Division of Labor, 50% share: jointly reviewed existing tutorials for NLTK, Downloading NLTK Data: text and coding, Basic NLTK Operations section: text and coding, Stopwords and Tokenizers: jointly worked on text and coding.

Introduction

This is our CS410 technical review for the Natural Language Toolkit (NLTK). Instead of just writing a PDF paper evaluating NLTK, we decided it would be useful if we went further, and not only explained and reviewed NLTK, but also developed an in-depth interactive tutorial in a Jupyter notebook for students interested in getting started with NLTK. In addition we have included links to books, papers and websites where appropriate to provide more detail on the theory underpinning the text processing concepts.

NLTK is a platform for writing Python programs for natural language processing (NLP) applications. This is a tutorial on some of the basic NLTK functionality and is meant as an introduction for beginners. However, the tutorial assumes you are familiar with programming in Python. If not, basic Python tutorials and installation instructions can be found here:

https://www.python.org/about/gettingstarted/

Version Notes:

- This tutorial was written in **Python 3.6**
- For the latest instructions on downloading and installing **NLTK version 3.4**, please see the official website: http://www.nltk.org/install.html.
- This tutorial accesses features that contained bugs in **NLTK v3.3**

This tutorial will cover the following topics:

- Installation
- Downloading NLTK Data
- Downloading NLTK Data
- Basic NLTK Operations
- Stopwords and Tokenizers
- Stemming and Lemmatization
- Parts of Speech Tagging
- Further Reading on NLTK

Installation

Option 1: Cloud

There are two options for installation. The first is not actually an installation, no files or data will be written to your local machine. This Jupyter notebook can be started directly from mybinder. It will take several minutes to start as it copies across file from this github repo, then builds and starts a docker container with the Python required libraries.

Option 2: Local Machine

The easiest way to install the prerequisites, if they are not already on your Windows, Mac or Linux machine, is to download and install Anaconda (Python version 3) from here, and then "conda install nltk", or refer to the official NLTK website

If you have an existing Python 3.5 or above installation and don't wish to install Anaconda, you can do the following, but you may need to be careful with versions:

```
pip install matplotlib jupyter nltk
```

Next, clone or download the GitHub repo from here. This contains the Jupyter notebook, **NLTK_Tutorial.ipynb**.

At a terminal/command line window you then type 'jupyter notebook' in the directory that contains the notebook. This will start the notebook server at port 8888 on your local machine and open a browser window. If you have any problems check this quickstart guide

Downloading NLTK Data

NLTK provides dozens of optional models, packages, grammars, and corpora (i.e. text collections) that you may choose to install.

To get started downloading the NLTK data, first import the nltk module:

In [ ]:

```
import nltk
```

To install the NLTK packages, pass a parameter to the download command specifying which package you want to download. For the purposes of this tutorial, you will only need to download the eight packages below:

In [ ]:

```
nltk.download('gutenberg')
nltk.download('masc_tagged')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')
nltk.download('tagsets')
nltk.download('universal_tagset')
```

NOTE: If you are running on your LOCAL machine and would like to download *all* the packages, you may specify the parameter, **all**. Uncomment and run the following command if you would like to download all packages.

In [ ]:

```
# WARNING: only uncomment the following command if you have 3+ GB available in local disk storage!
#nltk.download("all")
```

Basic NLTK Operations

The first collection we downloaded above, *gutenberg* is a small selection of texts from Project Gutenberg, which is an electronic archive project that has tens of thousands of books online. The *gutenberg* corpus is defined as *Plaintext Corpora* in NLTK, which is exactly as it sounds, non-tagged plain text.

The second collection we downloaded above, *masc_tagged* is a tagged corpus from the *Manually Annotated Sub-Corpus (MASC)* project. MASC is a subset of hundreds of thousands of English words from written works and transcribed speech. The MASC corpus defined as *Tagged Corpora* in NLTK and is annotated with different tokenizations, parts of speech tags, noun/verb shallow parsing, etc.

There are many other corpus formats and corpus readers classes defined in the package *nltk.corpus*. For more detailed information on the available NLTK corpora and corpus reader classes see: http://www.nltk.org/howto/corpus.html#corpus-reader-classes

To see the texts available from the *gutenberg* corpus:

In [ ]:

```
from nltk.corpus import gutenberg
gutenberg_fileids = gutenberg.fileids()
print(gutenberg_fileids)
```

The above is a list of the names of the books we can now access. We will select 'Alice in Wonderland' and open it with the NLTK plain text corpus reader class method **words()**. This reader class provides different data access methods to access the data in the *gutenberg* corpus, such as:

gutenberg corpus, such as:

- words()
- sents()
- paras()
- raw()

For detailed information on the NLTK corpus reader classes, see: http://www.nltk.org/api/nltk.corpus.reader.html#module-nltk.corpus.reader

Accessing and counting words

Let's first access the words in 'Alice in Wonderland':

In [ ]:

```
# get the number of words for 'Alice in Wonderland'
alice_words = gutenberg.words("carroll-alice.txt")
alice_nwords = len(alice_words)
print(alice_nwords)
```

To access the first and last word of the collection of words, simply use the appropriate index:

In [ ]:

```
print (alice_words[0])
print (alice_words[alice_nwords - 1])
```

We see that our concept of "word" might be different than NLTK's. A word here can be any sequence of non-space characters, or a punctuation symbol. Let's look at the first 20 words":

In [ ]:

```
print(alice_words[1:20])
```

To find the number of unique keywords in the word collection, use the **set** command:

In [ ]:

```
set(alice_words)
```

To obtain the count of unique words, use the **len** command:

In [ ]:

```
initial_unique_word_count = len(set(alice_words))
print (initial_unique_word_count)
```

The number of unique words here is overestimated because we haven't considered upper and lower case. Let's see if the number changes when we force all the text to lowercase and then count:

In [ ]:

```
alice_words_LC = [word.lower() for word in alice_words]
actual_unique_word_count = len(set(alice_words_LC))
print(actual_unique_word_count)
```

If you want to count the number of specific word occurences, you can use the **count** function:

In [ ]:

```
alice_words.count('Alice')
```

In addition to parsing the text into words, the plain text corpus reader class also provides a method for accessing the sentences in the text, using the **sent** function:

In [ ]:

```
alice_sentences = gutenberg.sents("carroll-alice.txt")

sentence_count = len(alice_sentences)
print("Number of sentences: {}".format(sentence_count))
```

To access any given sentence in the text, just index the array. For example, to access the 105th sentence:

In [ ]:

```
print(alice_sentences[104])
```

As an example of utility of the **sent** function, we could calculate the average number of words in each sentence: (keeping in mind this includes punctuation)

In [ ]:

```
print("Number of words per sentence: {}".format(alice_nwords/sentence_count))
```

Searching text

If you want to see the specific occurences within the text, along with some context in which the word appears, you can you the **concordance()** function, but first we must import the NLTK **Text** package and convert our words into an NLTK **Text** object. Note that **concordance()** is not case sensitive.

In [ ]:

```
from nltk.text import Text
textList = Text(alice_words)
textList.concordance("drink")
```

Try **concordance()** with 'Alice' as the parameter:

In [ ]:

```
textList.concordance('Alice')
```

It will only return the top 25 matches. If you want to see all the matches, you have to use the **lines** parameter:

Note that the **lines** parameter function did not work for values larger than the default size of 25 in NLTK v3.3 and was fixed in v3.4.

In [ ]:

```
textList.concordance('Alice', lines=1000)
```

To see a plot of the word offset certain words occur in the text

In [ ]:

```
%matplotlib inline
textList.dispersion_plot(["Alice","drink"])
```

The **ConcordanceIndex** class is closely related to the **Concordance** class and provides additional functionality for obtaining the indices of the the search word:

In [ ]:

```
from nltk.text import ConcordanceIndex
conIndex = ConcordanceIndex(alice_words)
conIndex.print_concordance("drink")
```

```
conIndex.print_concordance('drink')
print(conIndex.offsets('drink'))
```

Word frequency

To access the word frequency data in the text, use the **FreqDist** class:

In [ ]:

```
from nltk.probability import FreqDist
fdist = nltk.FreqDist(alice_words_LC)
```

An easy way to find the frequent words in a text is to call the **most_common** function and pass in N:

In [ ]:

```
fdist.most_common(25)
```

If you want to find a specific word in the frequency distribution, just pass it as an text index:

In [ ]:

```
print("rabbit appears {} times".format(fdist['rabbit']))
```

Access some more **FreqDist** methods:

In [ ]:

```
print("The word appearing the most times is {}".format(fdist.max()))
print("The number of words in the distribution is {}".format(fdist.N()))
print("The number of unique words in the distribution is {}".format(fdist.B()))
print("The frequency of the word \'alice\' is {}".format(fdist.freq('alice')))
print("The frequency of the word \'drink\' is {}".format(fdist.freq('drink')))
```

To plot the cumulative word distribution:

In [ ]:

```
fdist.plot(25, cumulative=True)
```

Stopwords and Tokenizers

Stopwords are high frequency, common words, such as "a", "an", "the", "from", "to", etc. Most of the time, NLP applications want to filter out (i.e. remove), stopwords before doing lexical/syntactic analysis because they have no added value to the analysis and can contribute to misleading results. For example, if you are comparing the similarity of two documents and don't remove stopwords, the similarity results will be misleading, showing a higher similarity simply due to matching common stopwords.

Stopwords

NLTK has a corpus of 2,400 stopwords in eleven language built into it. To include it and load the English language stopwords:

In [ ]:

```
from nltk.corpus import stopwords

english_stopwords = stopwords.words('english')
print (english_stopwords)
print("\nNumber of English stopwords is {}".format(len(english_stopwords)))
```

As an illustration of language differences, German has 30% more stopwords than English:

In [ ]:

```
german_stopwords = stopwords.words('german')
print(german_stopwords)
print("\nNumber of German stopwords is {}".format(len(german_stopwords)))
```

Remove the English stopwords from our lower case word list:

In [ ]:

```
alice_words_new = [word for word in alice_words_LC if word not in english_stopwords]

print("Number of words in \'Alice in Wonderland\' = {}".format(len(alice_words_LC)))
print("Number of words remaining after stopwords removed = {}".format(len(alice_words_new)))
print("Percentage of the text that consisted of stopwords =
{0:.2f}%".format(100*((len(alice_words_LC)-len(alice_words_new))/len(alice_words_LC))))
```

In [ ]:

```
fdist = nltk.FreqDist(alice_words_new)
print (fdist)
fdist.plot(25, cumulative=True)
```

In [ ]:

```
fdist.most_common(25)
```

Notice that after the stopwords are removed, many of the top words are now punctuation symbols. We could use Python to parse off words consisting solely of non-alphabetic characters and rebuild our list, but a quick and dirty way of keeping only "interesting" words, is to simply discard any word less than 3 characters and then look at the frequency distribution again.

In [ ]:

```
alice_words_new_truncated = [word for word in alice_words_new if word  if len(word) > 2 ]

fdist = nltk.FreqDist(alice_words_new_truncated)
fdist.plot(25, cumulative=True)
```

In [ ]:

```
fdist.most_common(25)
```

For more information on stopwords, see https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html

Tokenizers

NLTK provides numerous powerful tokenizers that can parse text using different algorithms dependent upon your specific application needs. For a full list and explaination of each NLTK tokenizer, see: http://www.nltk.org/api/nltk.tokenize.html

For a more in-depth look at the tokenization process, see:
https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en

For a detailed look at the complications of multi-word tokenization (such as "hot dog" and "red tape"), see this research paper:
https://d-nb.info/1046313010/34

For a comparison of NLTK to other open source tokenizers, see this paper: https://d-nb.info/1046313010/34

In this example, we show three tokenizers--

- **RegexpTokenizer**: a regular expression tokenizer
- **TweetTokenizer**: a tokenizer specifically designed to handle tweets from Twitter
- **word_tokenizer**: NLTK's recommended tokenizer for basic textual parsing

In [ ]:

```
from nltk.tokenize import RegexpTokenizer
from nltk.tokenize import TweetTokenizer
from nltk import word_tokenize
```

```
from nltk import word_tokenize

tweetTokenizer = TweetTokenizer()
regexpTokenizer = RegexpTokenizer(r'\w+')

text = "I have a secret.  I can't tell you.  #keepingQuiet"

tweetTokens = tweetTokenizer.tokenize(text)
regexpTokens = regexpTokenizer.tokenize(text)
wordTokens = word_tokenize(text)

print("Tweet tokens  {}".format(tweetTokens))
print("Regexp tokens {}".format(regexpTokens))
print("Word tokens   {}".format(wordTokens))
```

Notice that the output is different for each tokenizer.

The **TweetTokenizer** leaves punctuation intact, as in the word *can't*, but recognizes the text *#keepingQuiet* as a Twitter hashtag, so does not delete the # nor separate it from the following text.

The **RegexpTokenizer** uses the regular expression **\w+**, which greedily matches alphanumeric characters up until it encounters a non-alphanumeric character. Notice that it deletes not only the . and the #, but also deletes the apostrophe in *can't*, leaving the tokens *can* and *t*. For a good summary of regular expressions, see: https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285

The **word_tokenize** tokenizer leaves the punctuation between words intact, and splits the word *can't* into two tokens, *ca* and *n't*. This tokenizer is NLTK's recommended tokenizer and is actually a combination of two other basic NLTK tokenizers, first applying the NLTK **PunktSentenceTokenizer** followed by the NLTK **TreebankWordTokenizer**. It applies more intelligence to the sentence and word parsing than a simple regular expression tokenizer is able to.

When we used the corpus reader method **alice_words = gutenberg.words("carroll-alice.txt")** earlier to parse the 'Alice in Wonderland' text, the **words()** method actually is a wrapper that calls this **word_tokenize** function.

To illustrate the differences between using this methodology versus a basic regular expression parser, we can read in the 'Alice' text using the **raw** function, parse it with the regular expression parser, convert to lowercase, remove stopwords and compare the results.

In [ ]:

```
alice_raw = gutenberg.raw("carroll-alice.txt")
tokenizer = RegexpTokenizer(r'\w+')
alice_toks = tokenizer.tokenize(alice_raw)
alice_toks_LC = [word.lower() for word in alice_toks]
alice_toks_new = [word for word in alice_toks_LC if word not in english_stopwords]

print("Number of unique words in our initial analysis:
{}".format(len(set(alice_words_new_truncated))))
print("Number of unique words using reg exp tokenizer: {}".format(len(set(alice_toks_new))))
```

So there is a slight difference in the number of words. Let's compare the top five words from each tokenizer:

In [ ]:

```
fdist_orig = nltk.FreqDist(alice_words_new_truncated)
fdist_regexp = nltk.FreqDist(alice_toks_new)
```

In [ ]:

```
print("Orig   {}".format(fdist_orig.most_common(5)))
print("Regex  {}".format(fdist_regexp.most_common(5)))
```

So the results were the same in our current application, but bear in mind for other applications, this might not be the case. After we discuss parts of speech tagging later in this tutorial, we will show how using these two different tokenizers can result in very different results. This point is discussed in the final note of this tutorial titled: *Tokenizer/POS caution*

Stemming and Lemmatization

Stemming is used to reduce words to their "stems", i.e. their root form, by removing/replacing suffixes. For example, reducing the words: *eating, eaten, eats* to their common root *eat*.

Lemmatization is a process of determining the lemma, or base form of the word. Whereas stemming can be cruder with suffixes simply removed leaving a stem that isn't a valid word (such as *neurology* being reduced to the stem *neurolog*), lemmatization is a more complex process where a dictionary is accesssed and grammatical rules applied to determine a valid base form for the word.

In NLTK, there are two stemmers, the Porter Stemmer and the Lancaster Stemmer. The Porter Stemmer is based on algorithm developed in 1980. The original paper can be found here: https://tartarus.org/martin/PorterStemmer/def.txt

The Lancaster algorithm was developed more recently, and in general is considered more "aggressive" than the Porter algorithm in terms of reducing words to shorter stems.

NLTK uses the WordNet Lemmatizer. WordNet is a large lexical database of English language words that can be found here: https://wordnet.princeton.edu/

For a more detailed examination of stemming algorithms, see: http://research.ijais.org/volume4/number3/ijais12-450655.pdf and http://www.informationr.net/ir/19-1/paper605.html#.XAoO_BMzai4

In [ ]:

```
from nltk.stem import WordNetLemmatizer

porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()
lemmatizer = WordNetLemmatizer()
```

In [ ]:

```
list1 = ["automation","automate","automates"]
[porter.stem(w) for w in list1]
```

In [ ]:

```
[lancaster.stem(w) for w in list1]
```

In [ ]:

```
[lemmatizer.lemmatize(w) for w in list1]
```

Note the stemmers both produced the same result, while the lemmatizer left the words intact.

It is easy to create another simple example where the stemmers differ:

In [ ]:

```
list2 = ["runs","running","runner"]
[porter.stem(w) for w in list2]
```

In [ ]:

```
[lancaster.stem(w) for w in list2]
```

In [ ]:

```
[lemmatizer.lemmatize(w) for w in list2]
```

In this example, the porter stemmer retained the word "runner", whereas the lancaster stemmer shortened it to the root "run".

The lemmatizer kept the word list intact, except it dropped the "s" in runs.

Similarly, the stemmers and lemmatizer produce different results when we apply the algorithms to our Alice in Wonderland token set:

In [ ]:

```
alice_porter = [porter.stem(word) for word in sorted(set(alice_toks_new))]
alice_lancaster = [lancaster.stem(word) for word in sorted(set(alice_toks_new))]
alice_lemma = [lemmatizer.lemmatize(word) for word in sorted(set(alice_toks_new))]
```

```
print(alice_lemma[1:10])
print(alice_porter[1:10])
print(alice_lancaster[1:10])

print("Original count of unique words: {}".format(len(set(alice_toks_new))))
print("Lemmatization stem count: {}".format(len(set(alice_lemma))))
print("Porter stem count: {}".format(len(set(alice_porter))))
print("Lancaster stem count: {}".format(len(set(alice_lancaster))))
```

Note that the first 10 words in the set are stemmed slightly differently and that the total number of unique stems differs, with the lancaster stemmer producing the fewest stems, as might be expected from both the algorithm description of being the most aggressive stemmer, and from observing our simple "runner" example above.

Parts of Speech Tagging

One common task in NLP is classifying words into their parts of speech (POS) (e.g. verbs, nouns, adjectives, etc.). This categorization process is commonly referred to as POS-tagging.

For a more detailed look at POS tagging in NLP, see: https://pdfs.semanticscholar.org/4119/324f5bdbbf6b620e90ea26f5e4d27e6b8de0.pdf and https://nlp.stanford.edu/pubs/CICLing2011-manning-tagging.pdf

NLTK provides functionality to tag plain text in addition to providing corpora that is already pre-tagged. In the first step of this tutorial we downloaded the MASC tagged corpora, which we will look at in this section.

Below is a simple example of POS tagging starting with plain text:

In [ ]:

```
from nltk import word_tokenize
example_text = word_tokenize("This is a sample sentence that has a few words in it")
nltk.pos_tag(example_text)
```

The output from the **pos_tag** function is a list of (word, POS) tuples. In the above example, the tuple **('is', 'VBZ')** means that the word 'is' is a verb, present tense, 3rd person singular. The default tagset abbreviations can be non-intuitive, before you become familiar with the tagset.

For an explanation of each of the tagged parts of speech, issue the following NLTK help command:

In [ ]:

```
nltk.help.upenn_tagset()
```

Instead of scrolling through the above list of POS tags, you can look up a specific POS tag by passing the tag abbreviation in as a parameter:

In [ ]:

```
nltk.help.upenn_tagset("NNP")
```

If your plain text is already tokenized, you can simply call the **pos_tag** function directly. For example, we can tag our original text of 'Alice in Wonderland' with the following:

In [ ]:

```
nltk.pos_tag(alice_words)
```

The NLTK tagging algorithms are not perfect, and obviously can make mistakes with ambiguous words, etc, which is why pre-tagged corpora can be useful to NLP projects. For example, the MASC tagged corpus is manually verfied for validity so it can be used as a baseline for testing POS NLP software.

To see all of the pre-tagged texts available in the MASC corpus:

In [ ]:

```
from nltk.corpus import masc_tagged

masc_fileids = masc_tagged.fileids()
print(masc_fileids)
```

For this tutorial we will select the first text, which is a tagged transcript of the second U.S. Presidential debate between Al Gore and George W. Bush which took place in October, 2000.

In [ ]:

```
gore_bush_words = masc_tagged.words('spoken/2nd_Gore-Bush.txt')

print("Transcript length: {}".format(len(gore_bush_words)))
```

Now we can call the method **tagged_words()** to obtain the pre-tagged list of word/POS tuples:

In [ ]:

```
gore_bush_tagged_words = masc_tagged.tagged_words('spoken/2nd_Gore-Bush.txt')

print(gore_bush_tagged_words[0:24])
```

We can also call the method **tagged_sents()** to obtain the pre-tagged list of sentences:

In [ ]:

```
gore_bush_tagged_sents = masc_tagged.tagged_sents('spoken/2nd_Gore-Bush.txt')
```

To access the list of word/POS tuples in the 100th sentence of the text, just index the list:

In [ ]:

```
gore_bush_tagged_sents[99]
```

We can also call the method **tagged_paras** to obtain the pre-tagged list of paragraphs:

In [ ]:

```
gore_bush_tagged_paras = masc_tagged.tagged_paras('spoken/2nd_Gore-Bush.txt')
```

Access tagged paragraph by index:

In [ ]:

```
gore_bush_tagged_paras[10]
```

We can look at the frequency distribution of each POS tag:

In [ ]:

```
tags1 = nltk.FreqDist(tag for (word, tag) in gore_bush_tagged_words)
tags1.most_common()
```

If you want a less in-depth analysis of POS tags, i.e. you don't care about person, number, or tense and just want to know noun, verb, adverb, etc., you can specify the tagset **universal** which has the following tags:

- VERB - verbs (all tenses and modes)
- NOUN - nouns (common and proper)
- PRON - pronouns
- ADJ - adjectives
- ADV - adverbs
- ADP - adpositions (prepositions and postpositions)

- ADP - adpositions (prepositions and postpositions)
- CONJ - conjunctions
- DET - determiners
- NUM - cardinal numbers
- PRT - particles or other function words
- X - other: foreign words, typos, abbreviations
- . - punctuation

In [ ]:

```
gore_bush_universal_tagged_words = masc_tagged.tagged_words('spoken/2nd_Gore-Bush.txt', tagset='universal')
tags2 = nltk.FreqDist(tag for (word, tag) in gore_bush_universal_tagged_words)
tags2.most_common()
```

In [ ]:

```
tags2.plot()
```

Tokenizer/POS caution

As mentioned in the *Tokenizer* section of this tutorial, **word_tokenize()** is the NLTK recommended tokenizer, although we showed that the **RegexpTokenizer** produced the same top word count results. But for different applications, these two different tokenizers will result in very different results. A simple example is shown below:

In [ ]:

```
regExpTokenizer = RegexpTokenizer(r'\w+')

text = "I don't have shoes."

regexp_tokens = regExpTokenizer.tokenize(text)
word_tokens = word_tokenize(text)

print("Regexp toks: {}".format(regexp_tokens))
print("Word toks:   {}".format(word_tokens))

print("\nRegexp tuples: {}".format(nltk.pos_tag(regexp_tokens)))
print("Word tuples:   {}".format(nltk.pos_tag(word_tokens)))
```

Notice how even in a very simple sentence, the parts of speech vary. The **word_tokenize()** method produces the tokenized output for *don't* as *do* and *n't*. The **pos_tag()** method recognizes *n't* as the adverb *not*, whereas the output from the regular expression tokenizer is not recognized similarly and instead recognizes *don* as a different verb altogether than *do* and has no concept of negation. To help analyze this final example further, look in detail at the help method for each POS:

In [ ]:

```
nltk.help.upenn_tagset("RB")
nltk.help.upenn_tagset("NNS")
nltk.help.upenn_tagset("VB")
nltk.help.upenn_tagset("VBP")
nltk.help.upenn_tagset(".")
```

Further Reading on NLTK

- http://www.nltk.org/py-modindex.html The NLTK Python Module index
- http://www.nltk.org/genindex.html The NLTK complete alphabetical index
- https://github.com/nltk/nltk/wiki The NLTK WIKI