

Chapter 12

Workshop#3 POS Tagging Using NLTK

(Hour 5–6)

12.1 Introduction

In Chap. 3, we studied basic concepts and theories related to Part-of-Speech (POS) and various POS tagging techniques. This workshop will explore how to implement POS tagging by using NLTK starting from a simple recap on tokenization techniques and two fundamental processes in word-level progressing: stemming and stop-word removal, which will introduce two types of stemming techniques: Porter Stemmer and Snowball Stemmer that can be integrated with WordCloud commonly used in data visualization followed by the main theme of this workshop with the introduction of PENN Treebank Tagset and to create your own POS tagger.

12.2 A Revisit on Tokenization with NLTK

Text sentences are divided into subunits first and map into vectors in most NLP tasks. These vectors are fed into a model to encode where output is sent to a downstream task for results. NLTK (NLTK 2022) provides methods to divide text into subunits as tokenizers. Twitter sample corpus is extracted from NLTK to perform tokenization (Hardeniya et al. 2016; Kedia and Rasu 2020; Perkins 2014) in procedures below (Albrecht et al. 2020; Antic 2021; Bird et al. 2009):

1. Import NLTK package.
2. Import Twitter sample data.
3. List out fields.
4. Get Twitter string list.
5. List out first 15 Twitters.
6. Tokenize twitter.

Let us start with the import of NLTK package and download Twitter samples provided by NLTK platform.

```
In[1]  # Import NLTK
import nltk

# Download twitter_samples
# nltk.download('twitter_samples')
```

Import twitter samples dataset as *twtr* and check file id using *fileids()* method:

```
In[2]  # Import twitter samples from NTLK corpus (twtr)
from nltk.corpus import twitter_samples as twtr
```

```
In[3]  # Display Field IDs
twtr.fileids()
```

```
Out[3] ['negative_tweets.json', 'positive_tweets.json', 'tweets.20150430-23406.json']
```

Review first 5 Twitter messages:

```
In[4]  # Assign sample twitters (stwtr)
stwtr = twtr.strings('tweets.20150430-223406.json')
```

```
In[5]  # Display the first 5 sample twitters
stwtr[:5]
```

```
Out[5] ['RT @KirkKus: Indirect cost of the UK being in the EU is estimated to be costing Bri
tain £170 billion per year! #BetterOffOut #UKIP',
'VIDEO: Sturgeon on post-election deals http://t.co/8TJwrpbm0Y',
'RT @LabourEoin: The economy was growing 3 times faster on the day David Cameron bec
ame Prime Minister than it is today.. #BBCqt http://t.co/...',
'RT @GregLauder: the UKIP east lothian candidate looks about 16 and still has an asn
addy http://t.co/7eIU0c5Fai',
'RT @thesundaypeople: UKIP's housing spokesman rakes in £800k in housing benefit fro
m migrants. http://t.co/GVwb9Rcb4w http://t.co/c1AZxcLh...']
```

Import word_tokenize method from NLTK, name as *w_tok* to perform tokeniza-
tion on 5th Twitter message:

```
In[6]  # Import NLTK word tokenizer
from nltk.tokenize import word_tokenize as w_tok
```

```
In[7] ▶ # tokenize stwtr[4]
w_tok(stwtr[4])

Out[7] ['RT', '@', 'thesundaypeople', ':', 'UKIP', '"s", 'housing', 'spokesman',
'rakes', 'in', '£800k', 'in', 'housing', 'benefit', 'from', 'migrants', ':',
'http', ':', '//t.co/GVwb9Rcb4w', 'http', ':', '//t.co/c1AZxcLh...']
```



NLTK offers tokenization for punctuation and spaces *wordpunct_tokenize*. Let's use the 5th Twitter message to see how it works.

```
In[8] ▶ from nltk.tokenize import wordpunct_tokenize as wp_tok
wp_tok(stwtr[4])

Out[8] ['RT', '@', 'thesundaypeople', ':', 'UKIP', '"', 's', 'housing', 'spokesman',
'rakes', 'in', '£', '800k', 'in', 'housing', 'benefit', 'from', 'migrants', ':',
'http', ':', '/', 't', ':', 'co', '/', 'GVwb9Rcb4w', 'http', ':', '/', 't', ':', 'co', '/',
'c1AZxcLh', '...']
```



It can also tokenize words between hyphens and other punctuations. Further, NLTK's regular expression (RegEx) tokenizer can build custom tokenizers:

```
In[9] ▶ # Import the RegEx tokenizer
from nltk import regexp_tokenize as rx_tok
rx_pattern1 = '\w+'
rx_tok(stwtr[4], rx_pattern1)

Out[9] ['RT', 'thesundaypeople', 'UKIP', 's', 'housing', 'spokesman',
'rakes', 'in', '800k', 'in', 'housing', 'benefit', 'from', 'migrants',
'http', 't', 'co', 'GVwb9Rcb4w', 'http', 't', 'co', 'c1AZxcLh']
```



A simple regular expression filtered out words with alphanumeric characters only, but not punctuations in previous code. Another regular expression can detect and filter out both words containing alphanumeric characters and punctuation marks in the following code:

12.3.2 Why Stemming?

There is needless to extract every single word in a document but only concept or notion they represent such as information extraction and topic summarization in NLP applications. It can save computational capacity and preserve overall meaning of the passage. Stemming technique is to extract the overall meaning or words' base form instead of distinct words.

Let us look at how to perform stemming on text data.

12.3.3 How to Perform Stemming?

NLTK provides practical solution to implement stemming without sophisticated programming. Let us try two commonly used methods: (1) Porter Stemmer and (2) Snowball Stemmer in NLP.

12.3.4 Porter Stemmer

Porter Stemmer is the earliest stemming technique used in 1980s. Its key procedure is to remove words common endings and parse into generic forms. This method is simple and used in many NLP applications effectively.

Import Porter Stemmer from NLTK library:

```
In[11] ▶ # Import PorterStemmer as p_stem  
from nltk.stem.porter import PorterStemmer as p_stem
```

Try to stem words like *computer*.

```
In[12] ▶ p_stem().stem("computer")
```

```
Out[12] 'comput'
```



PorterStemmer simply removes suffix *-er* when processing *computer* to acquire *compute* which is incorrect. Hence this stemmer is basic.

Next, try to stem *dogs* to see what happens.

```
In[13] ▶ p_stem().stem("dogs")
```

```
Out[13] 'dog'
```



For the above code, *dogs* are converted from plural to singular, remove suffix *-s* and convert to *dog*.

Let's try more, say *traditional*.

```
In[14] ▶ p_stem().stem("traditional")
```

```
Out[14] 'tradi'
```



Stemmer may output an invalid word when dealing with special words e.g. *tradi* is acquired if suffix *-ional* is removed. *tradi* is not a word in English, it is a root form.

Let's work on words in plural form. There are 26 words extracted from a – z in plural form to perform PorterStemming:

```
In[15] ▶ # Define some plural words
```

```
w_plu = ['apes', 'bags', 'computers', 'dogs', 'egos', 'frescoes', 'generous', 'hats', 'igloos', 'jungles', 'kites', 'learners', 'mice', 'natives', 'openings', 'photos', 'queries', 'rats', 'scenes', 'trees', 'utensils', 'veins', 'wells', 'xylophones', 'yoyos', 'zens']
```

```
In[16] ▶ from nltk.stem.porter import PorterStemmer as p_stem
```

```
w_sgl = [p_stem().stem(wplu) for wplu in w_plu]
print(' '.join(w_sgl))
```

```
Out[16] ape bag comput dog ego fresco gener hat igloo jungl kite learner mice nativ
open photo queri rat scene tree utensil vein well xylophon yoyo zen
```



Porter Stemming will remove suffixes *-s* or *-es* to extract root form, that may result in single form such as *apes*, *bags*, *dogs*, etc. But in some cases, it will generate non-English words such as *gener*, *jungl* and *queri*.



Workshop 3.1 Try to stem a paragraph from The Adventures of Sherlock Holmes

1. Read Adventures_Holmes.txt *text file from The Adventures of Sherlock Holmes* (Doyle 2019; Gutenberg 2022)
2. Save contents into a string object "holmes_doc"
3. Extract a paragraph and tokenize it
4. Use Porter Stemming and output a list of stemmed words.

12.3.5 Snowball Stemmer

Snowball Stemmer provides improvement in stemming results as compared with Porter Stemmer and provides multi-language stemming solution. One can check languages using `languages()` method. Import from NLTK package to invoke Snowball Stemmer:

```
In[17] ➤ # Import Snowball Stemmer as s_stem
from nltk.stem.snowball import SnowballStemmer as s_stem
```

Review what languages Snowball stemmer can support:

```
In[18] ➤ # Display the s_stem language set
print(s_stem.languages)

Out[18] ('arabic', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian',
        'italian', 'norwegian', 'porter', 'portuguese', 'romanian', 'russian', 'spanish',
        'swedish')
```


Snowball Stemmer provides a variety of solutions in commonly used languages from Arabic to Swedish.

```
In[19] ➤ # Import Snowball Stemmer as s_stem and assign to English language
from nltk.stem.snowball import SnowballStemmer as s_stem
s_stem_ENG = s_stem(language="english")
```

Use same list of plural words (`w_plu`) to check how it works in Snowball Stemmer for comparison:

```
In[20] ➤ # Display the list of plural words
w_plu
```

Out[20] ['apes', 'bags', 'computers', 'dogs', 'egos', 'frescoes', 'generous', 'hats', 'igloos', 'jungles', 'kites', 'learners', 'mice', 'natives', 'openings', 'photos', 'queries', 'rats', 'scenes', 'trees', 'utensils', 'veins', 'wells', 'xylophones', 'yoyos', 'zens']

In[21]  *# Apply Snowball Stemmer onto the plural words*
 sgls = [s_stem_ENG.stem(wplu) for wplu in w_plu]
 print(' '.join(sgls))

Out[21] ape bag comput dog ego fresco generous hat igloo jungl kite learner mice
 nativ open photo queri rat scene tree utensil vein well xylophon yoyo zen

Try to compare with previous stemmer. What are the differences?



1. Snowball Stemmer achieved similar results as Porter Stemmer in most cases except in *generously* where Snowball Stemmer came up with a meaningful root form *generous* instead of *gener* in Porter Stemmer
2. Try some plural words to compare performance between Porter Stemmer vs Snowball Stemmer

12.4 Stop-Words Removal with NLTK

12.4.1 What Are Stop-Words?

There are input words and utterances to filter out impractical stop-words in NLP preprocessing such as: *a*, *is*, *the*, *of*, etc.

NLTK already provides a built-in stop-words package for this function. Let us see how it works.

12.4.2 NLTK Stop-Words List

Import stopwords module and call stopwords.words() method to list out all stop-words in English.

In[22] ▶

Import NLTK stop-words as wstops

```
from nltk.corpus import stopwords as wstops
print(wstops.words('english'))
```

Out[22]

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'you\'re', 'you\'v
e', 'you\'ll', 'you\'d', 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',
'himself', 'she', 'she\'s', 'her', 'hers', 'herself', 'it', 'it\'s', 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'thi
s', 'that', 'that\'ll', 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'bee
n', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an',
'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by',
'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before',
'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'ove
r', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'w
hy', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'su
ch', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's',
't', 'can', 'will', 'just', 'don', 'don\'t', 'should', 'should\'ve', 'now', 'd', 'll',
'm', 'o', 're', 've', 'y', 'ain', 'aren', 'aren\'t', 'couldn', 'couldn\'t', 'didn', 'di
dn\'t', 'doesn', 'doesn\'t', 'hadn', 'hadn\'t', 'hasn', 'hasn\'t', 'haven', 'haven\'t', 'i
sn', 'isn\'t', 'ma', 'mightn', 'mightn\'t', 'mustn', 'mustn\'t', 'needn', 'needn\'t', 'sh
an', 'shan\'t', 'shouldn', 'shouldn\'t', 'wasn', 'wasn\'t', 'weren', 'weren\'t', 'won',
'won\'t', 'wouldn', 'wouldn\'t']
```



1. Stop-words corpus size is not large.
2. All stop-words are commonly used in many documents. They effect storage and system efficiency in NLP applications if they are not removed.
3. This stop-word corpus is incomplete and subjective. There may be words considered as stop-words not included in this databank.

Use `stopwords.fileids()` function to review how many languages library of stop-words NLTK contains.

In[23] ▶

Import NLTK stop-words as wstops and display the FILE_IDS

```
from nltk.corpus import stopwords as wstops
print(wstops.fileids())
```

Out[23]

```
['arabic', 'azerbajjani', 'bengali', 'danish', 'dutch', 'english', 'finnish', 'french',
'german', 'greek', 'hungarian', 'indonesian', 'italian', 'kazakh', 'nepali',
'norwegian', 'portuguese', 'romanian', 'russian', 'slovene', 'spanish',
'swedish', 'tajik', 'turkish']
```

12.4.3 Try Some Texts

The above list shows all stop-words. Let us use a simple utterance:

```
In[24] ▶ # Import NLTK stop-words as wstops
from nltk.corpus import stopwords as wstops
wstops_ENG = wstops.words('english')
utterance = "Try to test for the stop word remove function to see how it
works."
utterance_clean = [w for w in utterance.split()
if w not in wstops_ENG]
```

Review results:

```
In[25] ▶ # Display the cleaned utterance
utterance_clean

Out[25] ['We', 'look', 'words', 'removed', 'text', 'following', 'code']
```



1. All commonly used stop-words such as *to*, *for*, *the*, *it*, are removed as shown in the example.
2. It has little effect to overall meaning of the utterance.
3. It requires same computational time and effort.

The following example uses *Hamlet* from *The Complete Works of Shakespeare* to demonstrate how stop-words are removed from text processing in NLP.

```
In[26] ▶ # Import the Gutenberg library from NLTK
from nltk.corpus import gutenberg as gub
hamlet = gub.words('shakespeare-hamlet.txt')
hamlet_clean = [w for w in hamlet if w not in wstops_ENG]
```

```
In[27] ▶ len(hamlet_clean)*100.0/len(hamlet)
```

```
Out[27] 69.26124197002142
```



This classic literature contains deactivated words. Nevertheless, these stop-words are unmeaningful in many NLP tasks that may affect results, so most of them are removed during pre-processing.

12.4.4 Create Your Own Stop-Words

Stop-word corpus can extract a list of string that can add any stop-words with simple `append()` function, but it is advisable to create a new stop-word library object name to begin.

12.4.4.1 Step 1: Create Own Stop-Word Library List

```
In[28] ➤ My_sws = wstops.words('english')
```

12.4.4.2 Step 2: Check Object Type and Will See It Has a Simple List

```
In[29] ➤ My_sws?
```

12.4.4.3 Step 3: Study Stop-Word List

```
In[30] ➤ My_sws
```

```
Out[30] ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
        "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he',
        'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
        'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
        'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', ... ]
```

12.4.4.4 Step 4: Add New Stop-Word "sampleSW" Using Append()

```
In[31] ➤ My_sws.append('sampleSW')
        My_sws[160:]
```

```
Out[31] ['ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't",
        'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren',
        "weren't", 'won', "won't", 'wouldn', "wouldn't", 'sampleSW']
```

Try this to see how it works.

```
In[32] ▶ # Import word_tokenize as w_tok
from nltk.tokenize import word_tokenize as w_tok

# Create the sample utterance
utterance = "This is a sample utterance which consits of eg as stop word
sampleSW."

# Tokenize the utterance
utt_toks = w_tok(utterance)

# Stop word removal
utt_nosw = [w for w in utt_toks if not w in My_sws]

# Display utterance without My stopwords
print(utt_nosw)

Out[32] ['This', 'sample', 'utterance', 'consits', 'eg', 'stop', 'word', '.']
```



Workshop 3.2 Stop-word Filtering on The Adventures of Sherlock Holmes

Use stop-word filtering technique for *The Adventures of Sherlock Holmes*:

1. Read Adventures_Holmes.txt text file
2. Save contents into a string object "holmes_doc"
3. Use stop-word technique just learnt to tokenize holmes_doc
4. Generate a list of word tokens with stop-words removed
5. Check any 3 possible stop-words to add into own stop-word list
6. Regenerate a new token list with additional stop-word removed

12.5 Text Analysis with NLTK

When text data has been processed and tokenized, basic analysis are required to calculate words or tokens, their distribution and usage frequency in NLP tasks. This allows understanding of main contents and topics accuracy in the document. Import a sample webtext (Firefox.txt) from NLTK library.

```
In[33] ▶ # Import webtext as wbtxt
from nltk.corpus import webtext as wbtxt

# Create sample webtext
wbtxt_s = wbtxt.sents('firefox.txt')
wbtxt_w = wbtxt.words('firefox.txt')

# Display total nos of webtext sentences in firefox.txt
len(wbtxt_s)
```

Out[33] 1138

Review the number of words as well.

```
In[34] ▶ # Display total nos of webtext words in firefox.txt
len(wbtxt_w)
```

Out[34] 102457



Firefox.txt contains sample texts extracted from Firefox discussion forum to serve as useful dataset for basic text-level analysis in NLP.

It can also obtain vocabulary size by passing through a set as shown in the following code:

```
In[35] ▶ # Define vocabulary object (vocab)
vocab = set(wbtxt_w)

# Display the size of Vocab
len(vocab)
```

Out[35] 8296

`nltk.FreqDist()` function is used to generate words frequency distribution occurred in the whole text as shown:

```
In[36] ▶ # Define Frequency Distribution object
fdist = nltk.FreqDist(wbtxt_w)
```

```
In[37] ▶ sorted(fdist, key=fdist.__getitem__, reverse=True)[0:30]
```

Out[37] ['.', 'in', 'to', 'the', 'not', 'when', 'on', 'a', 'is', 't', 'and', 'of', '(', 'page', 'for', 'with', ')', 'window', 'Firefox', 'does', 'from', 'open', '-', 'menu', 'should', 'bar', 'tab']

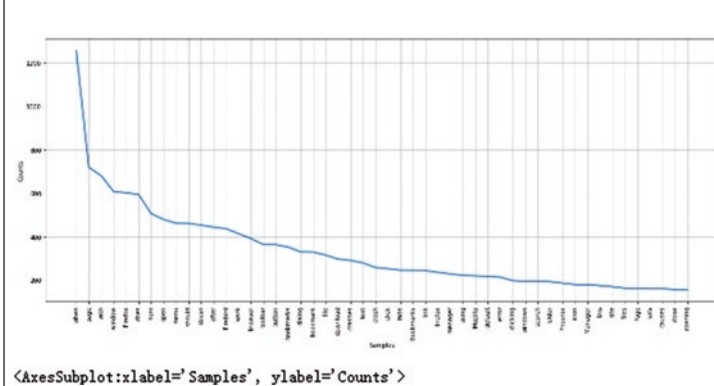


The above code generates top 30 frequently used words and punctuations in the whole text. *in*, *to* and *the* are top 3 on the list like other literatures as Firefox.txt text is the collection of users' discussion messages and contents about Firefox browser like conversations.

To exclude stop-words such as *the*, and *not*, use the following code to see f words frequency distribution longer than 3.

```
In[38] # Import Matplotlib pyplot object
import matplotlib.pyplot as plt
plt.figure(figsize=(20, 8))
lwords = dict([(k,v) for k,v in fdist.items() if len(k)>3])
fdist = nltk.FreqDist(lwords)
fdist.plot(50,cumulative=False)
```

Out[38]



Exclude stop-words such as *the*, *and*, *is*, and create a tuple dictionary to record words frequency. Visualize and transform them into a NLTK frequency distribution graph based on this dictionary as shown above.



Workshop 3.3 Text Analysis on The Adventures of Sherlock Holmes

1. Read Adventures_Holmes.txt text file
2. Save contents into a string object "holmes_doc"
3. Use stop-word technique from tokenize holmes_doc
4. Generate a word tokens list with stop-words removed
5. Use the technique learnt to plot first 30 frequently occurred words from this literature
6. Identify any special pattern related to word distribution. If no, try first 50 ranking words

12.6 Integration with WordCloud

See Fig. 12.2.

12.6.1 What Is WordCloud?

Wordcloud, also known as tag cloud, is a data visualization method commonly used in many web statistics and data analysis scenarios. It is a graphical representation of all words and keywords in sizes and colors. A word has the largest and bold in word cloud means it occurs frequently in the text (dataset).



Fig. 12.2 A sample WordCloud

12.7 POS Tagging with NLTK

The earlier part of this workshop had studied several NLP preprocessing tasks: tokenization, stemming, stop-word removal, word distribution in text corpus and data visualization using WordCloud. This section will explore POS tagging in NLTK.

12.7.1 What Is POS Tagging?

Part-of-Speech (POS) refers to words categorization process in a sentence/utterance into specific syntactic or grammatical functions.

There are 9 major POS in English: Nouns, Pronouns, Adjectives, Verbs, Prepositions, Adverbs, Determiners, interjection, and Conjunctions. POS tagging is to assign POS tags into each word token in the sentence/utterance.

NLTK supports commonly used tagset such as PENN Treebank (Treebank 2022) and Brown corpus. It allows to create own tags used for specific NLP applications.

12.7.2 Universal POS Tagset

A tagset consists of 12 universal POS categories which is constructed to facilitate future requirements for unsupervised induction of syntactic structure. When is combined with original treebank data, this universal tagset and mapping produce a dataset consisting of common POS in 22 languages (Albrecht et al. 2020; Antic 2021; Bird et al. 2009).

Figure 12.3 shows a table of universal POS tagset in English.

| Tag | Meaning | English Examples |
|------|---------------------|---|
| ADJ | adjective | <i>new, good, high, special, big, local</i> |
| ADP | adposition | <i>on, of, at, with, by, into, under</i> |
| ADV | adverb | <i>really, already, still, early, now</i> |
| CONJ | conjunction | <i>and, or, but, if, while, although</i> |
| DET | determiner, article | <i>the, a, some, most, every, no, which</i> |
| NOUN | noun | <i>year, home, costs, time, Africa</i> |
| NUM | numeral | <i>twenty-four, fourth, 1991, 14:24</i> |
| PRT | particle | <i>at, on, out, over per, that, up, with</i> |
| PRON | pronoun | <i>he, their, her, its, my, I, us</i> |
| VERB | verb | <i>is, say, told, given, playing, would</i> |
| . | punctuation | <i>. , ; !</i> |
| X | marks | |
| | other | <i>ersatz, esprit, dunno, gr8, univeristy</i> |

Fig. 12.3 Table of Universal POS Tagset in English

12.7.3 *PENN Treebank Tagset (English and Chinese)*

English Penn Treebank Tagset is used with English corpora developed by Prof. Helmut Schmid in TC project at the Institute for Computational Linguistics of the University of Stuttgart (TreeBank 2022). Figure 12.4 shows an original 45 used Penn Treebank Tagset.

A recent version of this English POS Tagset can be found at Sketchengine.eu (Sketchengine 2022a) and Chinese POS Tagset (Sketchengine 2022b).

NLTK provides direct mapping from tagged corpus such as Brown Corpus (NLTK 2022) to universal tags for implementation, e.g. tags VBD (for past tense verb) and VB (for base form verb) map to VERB only in universal tagset.

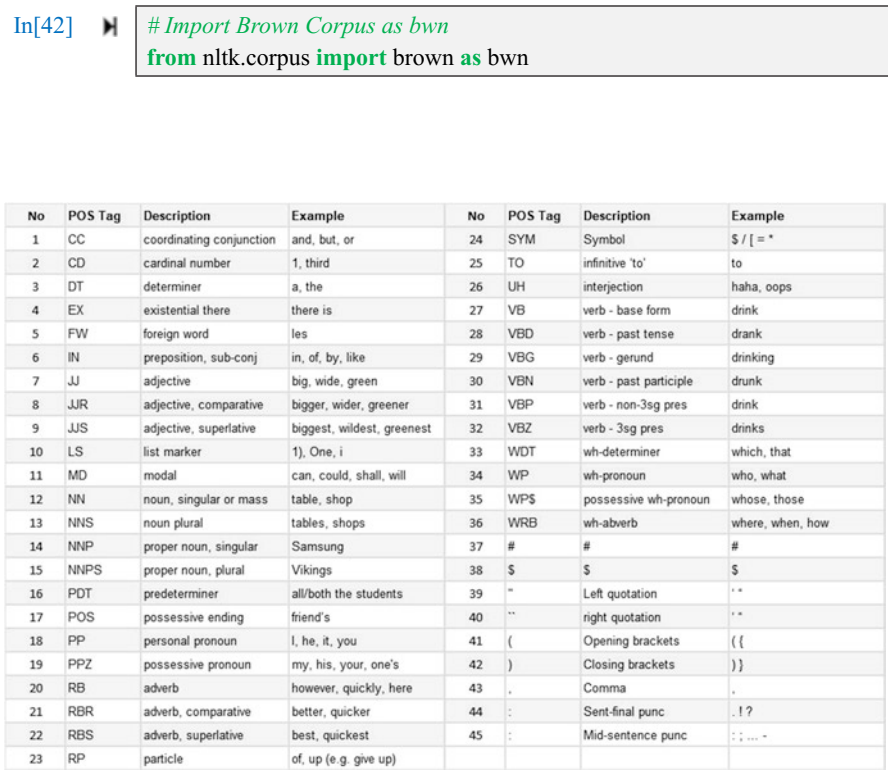


Fig. 12.4 Original 45 used Penn Treebank Tagset

```
In[43] ▶ bwn.tagged_words()[0:40]
Out[43] [('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'),
('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'),
('investigation', 'NN'), ('of', 'IN'), ('Atlanta's', 'NPS'), ('recent', 'JJ'),
('primary', 'NN'), ('election', 'NN'), ('produced', 'VBD'), (''', ''),
('no', 'AT'), ('evidence', 'NN'), ('''', '''), ('that', 'CS'), ('any', 'DT'),
('irregularities', 'NNS'), ('took', 'VBD'), ('place', 'NN'), ('.', '.'),
('The', 'AT'), ('jury', 'NN'), ('further', 'RBR'), ('said', 'VBD'),
('in', 'IN'), ('term-end', 'NN'), ('presentments', 'NNS'), ('that', 'CS'),
('the', 'AT'), ('City', 'NN-TL'), ('Executive', 'JJ-TL'),
('Committee', 'NN-TL'), ('.', '.'), ('which', 'WDT'), ('had', 'HVD')]
```



Fulton is tagged as NP-TL in example code above, a *proper noun* (NP) appears in a title (TL) context in Brown corpus that mapped to *noun* in universal tagset. These subcategories are to be considered instead of generalized universal tags in NLP application

12.7.4 Applications of POS Tagging

POS tagging is commonly used in many NLP applications ranging from Information Extraction (IE), Named Entity Recognition (NER) to Sentiment Analysis and Question-&-Answering systems.

Try the following and see how it works:

```
In[44] ▶ # Import word_tokenize and pos_tag as w_tok and p_tag
from nltk.tokenize import word_tokenize as w_tok
from nltk import pos_tag as p_tag

# Create and tokenizer two sample utterances utt1 and utt2
utt1 = w_tok("Give me a call")
utt2 = w_tok("Call me later")
```

Review these two utterances' POS tags:

```
In[45] ▶ p_tag(utt1, tagset='universal')
Out[45] [('Give', 'VERB'), ('me', 'PRON'), ('a', 'DET'), ('call', 'NOUN')]
```

In[46] ▶

p_tag(utt2, tagset='universal')

Out[46]

[('Call', 'VERB'), ('me', 'PRON'), ('later', 'ADV')]



- 1. The word *call* is a noun in text 1 and a verb in text 2.
- 2. POS tagging is used to identify a person, a place, or a location, based on the tags in NER.
- 3. NLTK also provides a classifier to identify such entities in text as shown in the following code:

In[47] ▶

utt_untag = w_tok("My dad was born in South America")
utt_untag

Out[47]

[('My', 'dad', 'was', 'born', 'in', 'South', 'America')]

In[48] ▶

utt_tagged = p_tag(utt_untag)
utt_tagged

Out[48]

[('My', 'PRP\$'), ('dad', 'NN'), ('was', 'VBD'), ('born', 'VBN'), ('in', 'IN'), ('South', 'NNP'), ('America', 'NNP')]

In[49] ▶

Import svgling package
import svgling

Import NLTK.ne_chunk as chunk
from nltk **import** ne_chunk **as** chunk

Display POS Tags chunk
chunk(utt_tagged)

Out[49]

```
graph TD
    S --> My
    S --> dad
    S --> was
    S --> born
    S --> in
    S --> GPE
    My --> PRP$
    dad --> NN
    was --> VBD
    born --> VBN
    in --> IN
    GPE --> South
    GPE --> America
    South --> NNP
    America --> NNP
```



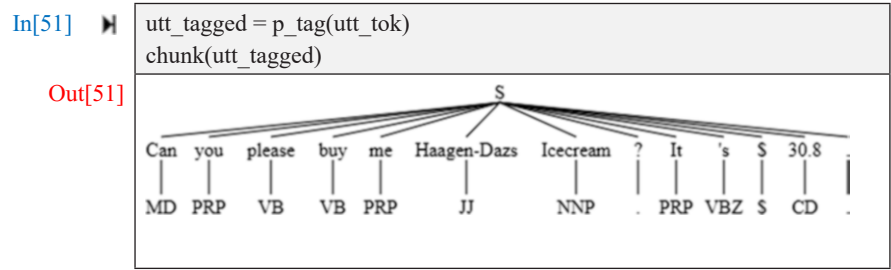
NLTK chunk() function is applied to NER to identify the chunker South America as a geopolitical entity (GPE) in this example. So far, there are examples using NLTK’s built-in taggers. Next section will look at how to develop own POS tagger.

In[50]

Try another example
utt_tok = w_tok("Can you please buy me Haagen-Dazs Icecream? It's \$30.8.")
print("Tokens are: ", utt_tok)

Out[50]

Tokens are: ['Can', 'you', 'please', 'buy', 'me', 'Haagen-Dazs', 'Icecream', '?', 'It', "'s", '\$', '30.8', '.']



- 1. The system treats '\$', '30.8', and '.' as separate tokens in this example. It is crucial because contractions have their own semantic meanings and own POS leading to the ensuing part of NLTK library POS tagger.
- 2. POS tagger in NLTK library outputs specific tags for certain words.
- 3. However, it makes a mistake in this example. Where is it?
- 4. Compare POS Tagging for the following sentence to identify problem. Explain.

In[52]

Try one more example
utt_tok = w_tok("Can you please buy me New-Zealand Icecream? It's \$30.8.")
print("Tokens are: ", utt_tok)
utt_tagged = nltk.pos_tag(utt_tok)
chunk(utt_tagged)

Out[52]

Tokens are: ['Can', 'you', 'please', 'buy', 'me', 'New-Zealand', 'Icecream', '?', 'It', "'s", '\$', '30.8', '.']

```
graph TD
    S[S] --- Can[Can]
    S --- you[you]
    S --- please[please]
    S --- buy[buy]
    S --- me[me]
    S --- New-Zealand[New-Zealand]
    S --- Icecream[Icecream]
    S --- question[?]
    S --- It[It]
    S --- s['s']
    S --- dollar[$]
    S --- 30.8[30.8]
    S --- period[.]
```



Workshop 3.5 POS Tagging on *The Adventures of Sherlock Holmes*

1. Read Adventures_Holmes.txt text file
2. Save contents into a string object "holmes_doc"
3. Extract three typical sentences from three stories of this literature
4. Use POS Tagging to these sentences
5. Use ne_chunk function to display POS tagging tree for these three sentences
6. Compare POS Tags among these example sentences and examine on how they work

12.8 Create Own POS Tagger with NLTK

This section will create own POS tagger using NLTK's tagged set corpora and sklearn Random Forest machine learning model.

The following example demonstrates a classification task to predict POS tag for a word in a sentence using NLTK treebank dataset for POS tagging, and extract word prefixes, suffixes, previous and neighboring words as features for system training.

Import all necessary Python packages as below:

```
In[53] # Import all necessary Python packages
import nltk
import numpy as np
from nltk import word_tokenize as w_tok
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.feature_extraction import DictVectorizer as DVect
from sklearn.model_selection import train_test_split as tt_split
from sklearn.ensemble import RandomForestClassifier as RFCClassifier
from sklearn.metrics import accuracy_score as a_score
from sklearn.metrics import confusion_matrix as c_matrix
```

```
In[54] # Define the ufeatures() class
def ufeatures(utt, idx):
    ftdist = {}
    ftdist['word'] = utt[idx]
    ftdist['dist_from_first'] = idx - 0
    ftdist['dist_from_last'] = len(utt) - idx
    ftdist['capitalized'] = utt[idx][0].upper() == utt[idx][0]
    ftdist['prefix1'] = utt[idx][0]
    ftdist['prefix2'] = utt[idx][:2]
    ftdist['prefix3'] = utt[idx][:3]
    ftdist['suffix1'] = utt[idx][-1]
    ftdist['suffix2'] = utt[idx][-2:]
    ftdist['suffix3'] = utt[idx][-3:]
    ftdist['prev_word'] = " if idx==0 else utt[idx-1]
    ftdist['next_word'] = " if idx==(len(utt)-1) else utt[idx+1]
    ftdist['numeric'] = utt[idx].isdigit()
    return ftdist
```

```
In[55] # Define the Retrieve Untagged Utterance (RUtterance) class
def RUtterance(utt_tagged):
    [utt,t] = zip(*utt_tagged)
    return list(utt)
```



Function `ufeatures()` converts input text into a dict object of features, whereas each utterance is passed with corresponding index of current token word which features are extracted. Let's use treebank tagged utterances with universal tags to label and train data:

```
In[56] utt_tagged = nltk.corpus.treebank.tagged_sents(tagset='universal')
```

```
In[57] utt_tagged
```

```
Out[57] [(('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), ('', '.'), ('will', 'VERB'), ('join', 'VERB'), ('the', 'DET'), ('board', 'NOUN'), ('as', 'ADP'), ('a', 'DET'), ('nonexecutive', 'ADJ'), ('director', 'NOUN'), ('Nov.', 'NOUN'), ('29', 'NUM'), ('', '.')), (('Mr.', 'NOUN'), ('Vinken', 'NOUN'), ('is', 'VERB'), ('chairman', 'NOUN'), ('of', 'ADP'), ('Elsevier', 'NOUN'), ('N.V.', 'NOUN'), ('', '.'), ('the', 'DET'), ('Dutch', 'NOUN'), ('publishing', 'VERB'), ('group', 'NOUN'), ('', '.')), ...]
```



1. In this example, universal tags are used for simplicity.
2. Of course, one can also use fine-grained treebank POS Tags for implementation.
3. Once done so, can now extract the features for each tagged utterance in corpus with training labels.

Use following code to extract the features:

In[58] ▶

```
# Define Extract Feature class (exfeatures)
def exfeatures(utt_tag):
    utt, tag = [], []

    for ut in utt_tag:
        for idx in range(len(ut)):
            utt.append(ufeatures(RUutterance(ut), idx))
            tag.append(ut[idx][1])

    return utt, tag
```

In[59] ▶

```
X,y = exfeatures(utt_tagged)
```

This example uses DVect to convert feature-value dictionary into training vectors. If the number of possible values for suffix3 feature is 40, there will be 40 features in output. Use following code to DVect:

In[60] ▶

```
# Define sample size
nsize = 10000

# Invoke Dict Vectorizer
dvect = DVect(sparse=False)

Xtran = dvect.fit_transform(X[0:nsize])
ysap = y[0:nsize]
```



This example has a sample size of 10,000 utterances which 80% of the dataset is used for training and other 20% is used for testing. RF (Random Forest) Classifier is used as POS tagger model as shown:

In[61] ▶

```
Xtrain,Xtest,ytrain,ytest = tt_split(Xtran, ysap, test_size=0.2,
    random_state=123)
```


In[62] ▶ `rfclassifier = RFClassifier(n_jobs=4)`
`rfclassifier.fit(Xtrain,ytrain)`

Out[62] `RandomForestClassifier(n_jobs=4)`



After system training, can perform POS Tagger validation by using some sample utterances. But before passing to `ptag_predict()` method, extract features are required by `ufeatures()` method as shown:

In[63] ▶ *# Define the POS Tags Predictor class (ptag_predict)*
`def ptag_predict(utt):`
 `utt_tagged = []`
 `fts = [ufeatures(utt, idx) for idx in range(len(utt))]`
 `fts = dvect.transform(fts)`
 `tgs = rfclassifier.predict(fts)`
 `return zip(utt, tgs)`



Convert utterance into corresponding features with `ufeatures()` method. The features dictionary extracted from this method is vectorized using previously trained `dvect`:

In[64] ▶ *# Test with a sample utterance (utt3)*
`utt3 = "It is an example for POS tagger"`
`for utt_tagged in ptag_predict(utt3.split()):`
 `print(utt_tagged)`

Out[64] `('It', 'PRON') ('is', 'VERB') ('an', 'DET') ('example', 'NOUN')`
`('for', 'ADP') ('POS', 'NOUN') ('tagger', 'NOUN')`



Use a sample utterance “utt3” and invoke `ptag_predict()` method to output tags for each word token inside `utt3` and review for accuracy afterwards.

In[65] ▶ `predict = rfclassifier.predict(Xtest)`

In[66] ▶ `a_score(ytest,predict)`

Out[66] `0.9355`

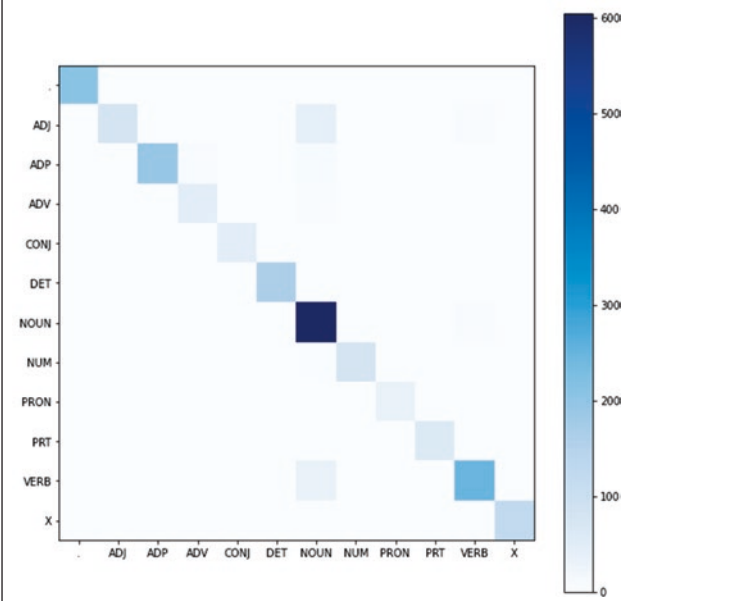


The overall `a_score` has approximately 93.6% accuracy rate and satisfactory. Next, let's look at confusion matrix (c-mat) to check how well can POS tagger perform.

In[67] `c_mat = c_matrix(ytest,predict)`

In[68] `pyplt.figure(figsize=(10,10))`
`pyplt.xticks(np.arange(len(rfclassifier.classes_),rfclassifier.classes_))`
`pyplt.yticks(np.arange(len(rfclassifier.classes_),rfclassifier.classes_))`
`pyplt.imshow(c_mat, cmap=pyplt.cm.Blues)`
`pyplt.colorbar()`

Out[68] `<matplotlib.colorbar.Colorbar at 0x1a7bf178ac0>`



Use classes from RF classifier as x and y labels to create a c-mat (confusion matrix). These labels are POS tags used for system training. The plot that follows shows a pictorial representation of confusion matrix.

Use classes from random forest classifier as x and y labels in the code for plotting confusion matrix.

It looks like the tagger performs relatively well for nouns, verbs, and determiners in sentences reflected in dark regions of the plot. Let’s look at some top features of the model from following code:

In[69] ▶

```
flist = zip(dvect.get_feature_names_out(),
rfclassifier.feature_importances_)
sfeatures = sorted(flist,key=lambda x: x[1], reverse=True)
print(sfeatures[0:20])
```

Out[69]

```
[('prefix1=*', 0.01879126572576589), ('capitalized',
0.016126122321655705), ('dist_from_last', 0.01331709807586213),
('prefix2=th', 0.012352866808396801), ('suffix2=ed',
0.012234201616289414), ('suffix2=', 0.012062684250210154),
('suffix2=he', 0.010422812160423846), ('suffix1=d',
0.010355753995257486), ('word=the', 0.010165678287431567),
('dist_from_first', 0.009695691000093641), ('word=',
0.009431485047631907), ('prefix1=', 0.009399515690975522),
('prefix1=t', 0.008812458552345161), ('next_word=',
0.008621099914970934), ('suffix1=s', 0.008390637059829514), ('word=-',
0.007823442817109226), ('numeric', 0.007360158530919634),
('suffix3=the', 0.0071258684674633845), ('suffix2=',
0.007118098327227603), ('word=of', 0.0070616788432973174)]
```



- 1. The RF feature importance is stored in Python feature_importances list. Some of the suffix features have higher importance scores than others
- 2. For instances, words ending with *-ed* are usually verbs in past tense which make sense in many situations, and punctuations like commas may affect POS tagging performance in some situations



- Workshop 3.6 Revisit POS Tagging on The Adventures of Sherlock Holmes with Additional Tagger**
- 1. Read Adventures_Holmes.txt text file
 - 2. Save contents into a string object "holmes_doc"
 - 3. Extract three typical sentences from three stories of this literature
 - 4. Use method learnt to create own POS taggers. What are new POS tags to add or use?
 - 5. Try new POS taggers for these three typical sentences and compare results with previous workshop

References

- Albrecht, J., Ramachandran, S. and Winkler, C. (2020) Blueprints for Text Analytics Using Python: Machine Learning-Based Solutions for Common Real World (NLP) Applications. O'Reilly Media.
- Antic, Z. (2021) Python Natural Language Processing Cookbook: Over 50 recipes to understand, analyze, and generate text for implementing language processing tasks. Packt Publishing.
- Bird, S., Klein, E., and Loper, E. (2009). Natural language processing with python. O'Reilly.
- Doyle, A. C. (2019) The Adventures of Sherlock Holmes (AmazonClassics Edition). AmazonClassics.
- Gutenberg (2022) Project Gutenberg official site. <https://www.gutenberg.org/> Accessed 16 June 2022.
- Hardeniya, N., Perkins, J. and Chopra, D. (2016) Natural Language Processing: Python and NLTK. Packt Publishing.
- Kedia, A. and Rasu, M. (2020) Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing.
- NLTK (2022) NLTK official site. <https://www.nltk.org/>. Accessed 16 June 2022.
- Perkins, J. (2014). Python 3 text processing with NLTK 3 cookbook. Packt Publishing Ltd.
- Sketchengine (2022a) Recent version of English POS Tagset by Sketchengine. <https://www.sketchengine.eu/english-treetagger-pipeline-2/>. Accessed 21 June 2022.
- Sketchengine (2022b) Recent version of Chinese POS Tagset by Sketchengine. <https://www.sketchengine.eu/chinese-penn-treebank-part-of-speech-tagset/>. Accessed 21 June 2022.
- Treebank (2022) Penn TreeBank Release 2 official site. <https://catalog.ldc.upenn.edu/docs/LDC95T7/treebank2.index.html>. Accessed 21 June 2022.