Portfolio 3: WordNet

Bushra Rahman

A Summary of WordNet

WordNet is a **lexical database** that organizes words by their hierarchical relationships. The words in WordNet are lemmatized and are classified by their part of speech: **nouns**, **verbs**, **adjectives**, and **adverbs**. Words that are synonyms of each other are grouped into **synsets**. Each synset is accompanied by a **gloss** (a short definition) and **examples** of its usage. Synsets are connected to each other through hierarchical relationships like **hypernyms** and **hyponyms**, **meronyms** and **holonyms**, and **troponyms**.

Exploring Noun Synsets

The following code outputs all synsets of the word **"language"**. The output is a list of synsets, where each synonym is given as *lemma.POS.num*. *POS* is the part of speech (eg. *n* for noun and *v* for verb), and *num* is the numerical listing of the lemma under its dictionary definition (eg. *01* is the 1st dictionary meaning).

In [1]:
```
from nltk.corpus import wordnet as wn  # give WordNet the simple alias wn
noun = 'language'
wn.synsets(noun)
```

Out[1]:
```
[Synset('language.n.01'),
 Synset('speech.n.02'),
 Synset('lyric.n.01'),
 Synset('linguistic_process.n.02'),
 Synset('language.n.05'),
 Synset('terminology.n.01')]
```

Functions that can be used on a synset include:

- definition() to retrieve its gloss
- examples() to retrieve its usage cases
- lemmas() to retrieve related lemmas that are synonyms

The synset chosen for these methods is **Synset('lyric.n.01')**, the third element from the list of synsets outputted above.

In [2]:
```
noun_syns = wn.synsets(noun)[2]  # save synset in variable
noun_syns.definition()
```

Out[2]:
```
'the text of a popular song or musical-comedy number'
```

In [3]:
```
noun_syns.examples()
```

```
Out[3]: ['his compositions always started with the lyrics',
         'he wrote both words and music',
         'the song uses colloquial language']
```

```
In [4]: noun_syns.lemmas()
```

```
Out[4]: [Lemma('lyric.n.01.lyric'),
         Lemma('lyric.n.01.words'),
         Lemma('lyric.n.01.language')]
```

All noun synsets in WordNet are hyponyms of the top-level word **"entity"**. This means that the hierarchy of hypernyms for every noun can be traced back up to "entity". By traversing up the hypernym hierarchy for a given noun, all of its hypernyms up until "entity" can be retrieved.

```
In [5]: hyper = noun_syns.hypernyms()[0]
        top = wn.synset('entity.n.01')
        while hyper:
            print(hyper)  # print hypernym synset
            if hyper == top:
                break
            if hyper.hypernyms():
                hyper = hyper.hypernyms()[0]
```

```
Synset('text.n.01')
Synset('matter.n.06')
Synset('writing.n.02')
Synset('written_communication.n.01')
Synset('communication.n.02')
Synset('abstraction.n.06')
Synset('entity.n.01')
```

The methods for **hypernyms**, **hyponyms**, **meronyms**, and **holonyms** output lists of synsets fitting those relationships for a given synset. The method for **antonyms**, however, can only be applied to a lemma, not a synset. So, a lemma first has to be extracted from the synset. Sometimes, no such synsets exist, and an empty list is outputted.

```
In [6]: noun_syns.hypernyms()
```

```
Out[6]: [Synset('text.n.01')]
```

```
In [7]: noun_syns.hyponyms()
```

```
Out[7]: [Synset('love_lyric.n.01')]
```

```
In [8]: noun_syns.part_meronyms()
```

```
Out[8]: []
```

```
In [9]: noun_syns.part_holonyms()
```

```
Out[9]: [Synset('song.n.01')]
```

```
In [10]: noun_syns.lemmas()[0].antonyms()
```

```
Out[10]: []
```

Exploring Verb Synsets

The following code outputs all synsets of the word **"speak"**.

```
In [11]: verb = 'speak'
         wn.synsets(verb)
```

```
Out[11]: [Synset('talk.v.02'),
          Synset('talk.v.01'),
          Synset('speak.v.03'),
          Synset('address.v.02'),
          Synset('speak.v.05')]
```

The same methods to retrieve the gloss, usage cases, and lemmas for nouns synsets can also be used on verb synsets. The synset chosen for these methods is **Synset('talk.v.01')**, the second element from the list of synsets outputted above.

```
In [12]: verb_syns = wn.synsets(verb)[1]  # save synset in variable
         verb_syns.definition()
```

```
Out[12]: 'exchange thoughts; talk with'
```

```
In [13]: verb_syns.examples()
```

```
Out[13]: ['We often talk business', 'Actions talk louder than words']
```

```
In [14]: verb_syns.lemmas()
```

```
Out[14]: [Lemma('talk.v.01.talk'), Lemma('talk.v.01.speak')]
```

While nouns in WordNet can be traced back up to one uniform top-level synset, verbs cannot. Instead, verb hypernyms become unrelated or repetitive the higher up in the hierarchy they are. In the code below, after a few iterations, the same synset "act.v.01" starts to repeat.

```
In [15]: hyper = verb_syns.hypernyms()[0]
         for i in range(5):
             print(hyper)  # print hypernym synset
             if hyper.hypernyms():
                 hyper = hyper.hypernyms()[0]
```

```
Synset('communicate.v.02')
Synset('interact.v.01')
Synset('act.v.01')
Synset('act.v.01')
Synset('act.v.01')
```

Using Morphy

Morphy finds the root form of a variety of different versions of the same word. Using morphy, different variations of "language" and "speak" can be traced back to their root forms.

```
In [16]: wn.morphy('languages', wn.NOUN)
```

```
Out[16]: 'language'
```

```
In [17]: wn.morphy('speaks', wn.VERB)
```

```
Out[17]: 'speak'
```

Word Similarity and Word Sense Disambiguation

Similar words in WordNet can be compared using the **Wu-Palmer similarity metric**, which gives words a similarity score between 0 (dissimilar) to 1 (identical). Word sense disambiguation (WSD), which is the task of identifying which particular synset of a word matches its use in a given sentence, can be done using the **Lesk Algorithm**. The algorithm uses contextual words to identify which synset matches the target word.

The output of *wn.synsets('language')* from above displayed a synset *Synset('speech.n.02')*. The output of *wn.synsets('speak')* from above displayed a synset *Synset('speak.v.05')*. How similar are these two words, "speech" and "speak"? According to the Wu-Palmer index, not very similar — their score is given as 0.22.

```
In [18]: wn.wup_similarity(wn.synset('speech.n.02'), wn.synset('speak.v.05'))
```

```
Out[18]: 0.2222222222222222
```

The word "talk" can be a verb or a noun depending on how it is used in a sentence. The Lesk Algorithm can be used to determine which synset of "talk" is used in each sentence.

```
In [19]: from nltk.wsd import lesk

sent1 = 'I attended your talk yesterday.'
sent2 = 'You talk too much.'
print(lesk(sent1, 'talk', 'n'))
print(lesk(sent2, 'talk', 'v'))
```

```
Synset('talk.n.03')
Synset('lecture.v.01')
```

SentiWordNet

SentiWordNet is a lexical resource built on top of WordNet for sentiment analysis and opinion mining. SentiWordNet associates a given synset with 3 numerical sentiment scores for **objectivity**, **positivity**, and **negativity**. SentiWordNet can be used to analyze tokenized text scraped from sources like Twitter, which holds plenty of linguistic data on personal opinions.

An example word for sentiment analysis is **"eloquent"**, an adjective to describe speech. "Eloquent" is typically used as a compliment, thus giving it a positive connotation. Does SentiWordNet reflect this? First, the senti-synset(s) for this word are output in list form. Because swn.senti_synsets() does not output synsets in a list like wn.synsets() did, the function has to be wrapped in a list.

In [20]:
```
from nltk.corpus import sentiwordnet as swn

senti = 'eloquent'
senti_syns = list(swn.senti_synsets(senti))
senti_syns
```

Out[20]:
```
[SentiSynset('eloquent.s.01')]
```

The following loop outputs the polarity scores for each senti-synset in the list. The output above shows that "eloquent" had only 1 senti-synset. The following code can be run using different synsets besides "eloquent" that may have more than 1 element.

In [21]:
```
# syns = list(swn.senti_synsets('eloquent'))[0]

for syns in senti_syns:
    print(syns)
    print('Positive score = ', syns.pos_score())
    print('Negative score = ', syns.neg_score())
    print('Objective score = ', syns.obj_score(), '\n')
```

```
<eloquent.s.01: PosScore=0.375 NegScore=0.125>
Positive score =  0.375
Negative score =  0.125
Objective score =  0.5
```

The output of the scores shows that "eloquent" is mostly designated as an **objective** word by SentiWordNet. It is, however, more positive than negative. The fact that the scores for positivity and negativity go up to 3 significant figures seems to imply that SentiWordNet could not interpret "eloquent" as an overwhelmingly positive or negative word. If the word had been more overwhelmingly positive or negative in connotation, then SentiWordNet might have been able to give it a simpler positive or negative score, like 0.5 or 1. The **utility** that these sentiment scores provide is in building NLP applications like chatbots, which need to understand human sentiment in language in order to converse realistically.

Collocations

A **collocation** is a grouping of words that usually occur together with an atypically high frequency. These words, when grouped, typically form common terms and phrases, and can't be replicated using synonyms. The following code outputs collocations for **text4**, the Inaugural corpus, in nltk.book.

In [22]:
```
from nltk.book import text4

print('\n')
text4.collocations()
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
United States; fellow citizens; years ago; four years; Federal
Government; General Government; American people; Vice President; God
bless; Chief Justice; one another; fellow Americans; Old World;
Almighty God; Fellow citizens; Chief Magistrate; every citizen; Indian
tribes; public debt; foreign nations
```

Mutual information, or point-wise mutual information (PMI), is given by the formula $log_2 \frac{P(x,y)}{P(x)*P(y)}$. The following code calculates the PMI of the collocation **"Indian tribes"** in the text:

In [23]:
```
import math

text = ' '.join(text4.tokens)  # create a text variable to contain text4
vocab = len(set(text4))
collocation = 'Indian tribes'

# P(x,y)
xy = text.count(collocation)/vocab
print('P(' + collocation + ') =', xy)

# P(x)
x = text.count(collocation.split()[0])/vocab
print('P(' + collocation.split()[0] + ') =', x)

# P(y)
y = text.count(collocation.split()[1])/vocab
print('P(' + collocation.split()[1] + ') =', y)

# PMI formula
pmi = math.log2(xy / (x * y))
print('PMI of \"' + collocation + '\" =', pmi)
```

```
P(Indian tribes) = 0.000598503740648379
P(Indian) = 0.0010972568578553616
P(tribes) = 0.000598503740648379
PMI of "Indian tribes" = 9.831882997592349
```

The PMI score for "Indian tribes" is about 9.83. This score is a positive value as opposed to negative or 0, thus indicating that *x* and *y* occur at a rate so frequent that it is not likely to be due to chance. Therefore, "Indian tribes" is a collocation.