Portfolio Assignment: WordNet

Cady Baltz (cmb180010)

2/26/2023

1. Summary of WordNet

WordNet is a lexical database of words that is organized hierarchically. Words are grouped with their synonyms into "synsets", which then form relationships with other synsets (such as part-whole relationships). This project was developed at Princeton University in the 1980s according to theories that humans organize concepts mentally in a hierarchy.

2. Select a noun and output all of its synsets

```
# Imports and downloads required for using WordNet

import nltk
nltk.download('wordnet')
nltk.download('omw-1.4')

from nltk.corpus import wordnet as wn
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
```

```
# Output all synsets (synonym sets) of my chosen noun (blade)

print("All synsets for the word 'blade':\n")
wn.synsets('blade')
```

```
All synsets for the word 'blade':

[Synset('blade.n.01'),
 Synset('blade.n.02'),
 Synset('blade.n.03'),
 Synset('sword.n.01'),
 Synset('blade.n.05'),
 Synset('blade.n.06'),
 Synset('blade.n.07'),
 Synset('blade.n.08'),
 Synset('blade.n.09')]
```

3. Analyze one synset from the list above

```
# Selected one synset from the list outputted above
blade_synset = wn.synset('blade.n.04')

# Extract its definition/gloss
print("Definition: " + str(blade_synset.definition()))
```

```
Definition: a cutting or thrusting weapon that has a long metal blade and a hilt with a hand guard
```

```
# Extract its usage examples (found none)
print("Usage examples: " + str(blade_synset.examples()))
```

```
Usage examples: []
```

```
# Extract its lemmas (WordNet entries that are synonyms in this set)
lemmas = blade_synset.lemmas()
print("Lemmas in the blade synset: \n")
lemmas
```

```
Lemmas in the blade synset:

[Lemma('sword.n.01.sword'),
 Lemma('sword.n.01.blade'),
```

```
 Lemma('sword.n.01.brand'),
 Lemma('sword.n.01.steel')]
```

```
# Traverse up the WordNet hierarchy (all the way up to entity.n.01), outputting synsets as you go

# Create a function "hyper" that returns the hypernyms of a synset
hyper = lambda s: s.hypernyms()

# Use the NLTK closure method to create a list of synsets for each level of hypernym
print("Traversing up the WordNet Hierarchy from blade.n.04 to entity.n.01:\n")
list(blade_synset.closure(hyper))
```

```
Traversing up the WordNet Hierarchy from blade.n.04 to entity.n.01:

[Synset('weapon.n.01'),
 Synset('instrument.n.01'),
 Synset('device.n.01'),
 Synset('instrumentality.n.03'),
 Synset('artifact.n.01'),
 Synset('whole.n.02'),
 Synset('object.n.01'),
 Synset('physical_entity.n.01'),
 Synset('entity.n.01')]
```

For nouns, the hierarchical structure of WordNet dictates that they all fall under one large synset, 'entity.n.01'. As you can see from the previous output, nouns are then divided into more and more specific groups. Overall, for the word 'blade', these divisions were very logical and helped narrow down the synset to 'weapon' right before 'blade', which would be a useful classification for this word.

4. Additional information about the synset's hierarchical relationships

```
# Extract its hypernyms (higher synsets than the given one)
print("Hypernyms:\n")

# Prints a more general synset for a blade (weapon)
blade_synset.hypernyms()
```

```
Hypernyms:

[Synset('weapon.n.01')]
```

```
# Extract its hyponyms (lower synsets than the given one)
print("Hyponyms:\n")

# Prints a list of more specific types of blades, such as broadsword
blade_synset.hyponyms()
```

```
Hyponyms:

[Synset('backsword.n.02'),
 Synset('broadsword.n.01'),
 Synset('cavalry_sword.n.01'),
 Synset('cutlas.n.01'),
 Synset('falchion.n.01'),
 Synset('fencing_sword.n.01'),
 Synset('rapier.n.01')]
```

```
# Extract its meronyms (synsets that are part of the given one)

# Found none
print("Member meronyms: " + str(blade_synset.member_meronyms()))

# Prints words that describe different "parts" of a blade, like a blade's hilt
print("Part meronyms: " + str(blade_synset.part_meronyms()))

# Found none
print("Substance meronyms: " + str(blade_synset.substance_meronyms()))
```

```
Member meronyms: []
Part meronyms: [Synset('blade.n.09'), Synset('foible.n.02'), Synset('forte.n.03'), Synset('haft.n.01'), Synset('hilt.n.01'), Synset('poi
Substance meronyms: []
```

```
# Extract its holonyms (synsets that make up the whole of which the current synset is a part of)

# Found none
print("Member meronyms: " + str(blade_synset.member_holonyms()))

# Found none
print("Part meronyms: " + str(blade_synset.part_holonyms()))

# Found none
print("Substance meronyms: " + str(blade_synset.substance_holonyms()))
```

```
Member meronyms: []
Part meronyms: []
Substance meronyms: []
```

```
# Extract its antonyms

# WordNet finds antonyms for specific lemmas, as opposed to synsets
for lemma in lemmas:

  # No antonyms were found for any of blade's lemmas
  print("Antonyms for " + str(lemma) + ": " + str(lemma.antonyms()))
```

```
Antonyms for Lemma('sword.n.01.sword'): []
Antonyms for Lemma('sword.n.01.blade'): []
Antonyms for Lemma('sword.n.01.brand'): []
Antonyms for Lemma('sword.n.01.steel'): []
```

5. Select a verb and output all synsets

```
# Output all synsets of my chosen verb

print("All synsets for the verb 'shatter':\n")
wn.synsets('shatter')
```

```
All synsets for the verb 'shatter':

[Synset('shatter.v.01'), Synset('shatter.v.02'), Synset('shatter.v.03')]
```

6. Analyze one synset from the list above

```
# Selected one synset from the list outputted above
shatter_synset = wn.synset('shatter.v.01')

# Extract its definition/gloss
print("Definition: " + str(shatter_synset.definition()))
```

```
Definition: break into many pieces
```

```
# Extract usage examples for the verb shatter
print("Usage examples: " + str(shatter_synset.examples()))
```

```
Usage examples: ['The wine glass shattered']
```

```
# Extract its lemmas
lemmas = shatter_synset.lemmas()
print("Lemmas in the shatter synset: \n")
lemmas
```

```
Lemmas in the shatter synset:

[Lemma('shatter.v.01.shatter')]
```

```
# Traverse up the WordNet hierarchy, outputting synsets as you go

# Create a function "hyper" that returns the hypernyms of a synset
hyper = lambda s: s.hypernyms()

# Use the NLTK closure method to create a list of synsets for each level of hypernym
print("Traversing up the WordNet Hierarchy for shatter:\n")
list(shatter_synset.closure(hyper))
```

```
Traversing up the WordNet Hierarchy for shatter:

[Synset('burst.v.08'),
 Synset('break.v.02'),
 Synset('change_integrity.v.01'),
 Synset('change.v.02')]
```

In comparison with nouns, I found that verbs were less logically well-organized in WordNet. When I traversed up the hierarchy for 'shatter', I reached a final synset of 'change', while all nouns arrive at the same final synset. Additionally, my verb did not have as many layers of hypernyms as my noun did, and they were less descriptive overall.

7. Use Morphy to find as many different forms of the word as you can

```
# Analyzing both words from earlier ('blade' and 'shatter') with Morphy

for word in ['blade', 'shatter']:
  print("Different forms of the word '" + word + "':")
  print("Adjective form: " + str(wn.morphy(word, wn.ADJ)))
  print("Verb form: " + str(wn.morphy(word, wn.VERB)))
  print("Adverb form: " + str(wn.morphy(word, wn.ADV)))
  print("Noun form: " + str(wn.morphy(word, wn.NOUN)))
  print()
```

```
Different forms of the word 'blade':
Adjective form: None
Verb form: None
Adverb form: None
Noun form: blade

Different forms of the word 'shatter':
Adjective form: None
Verb form: shatter
Adverb form: None
Noun form: None
```

8. Compare two words that I think might be similar

```
# Find the specific synsets I am interested in

# First, I printed all of the synsets
print("All synsets for the word 'happiness':")
print(wn.synsets('happiness'))
print()
print("All synsets for the word 'joy':")
print(wn.synsets('joy'))
print()

# After analyzing the definitions of each synset, I chose the following two synsets for comparison

happiness_synset = wn.synset('happiness.n.01')
print("'happiness' definition: " + str(happiness_synset.definition()))

joy_synset = wn.synset('joy.n.01')
print("'joy' definition: " + str(joy_synset.definition()))
```

```
All synsets for the word 'happiness':
[Synset('happiness.n.01'), Synset('happiness.n.02')]

All synsets for the word 'joy':
[Synset('joy.n.01'), Synset('joy.n.02'), Synset('rejoice.v.01'), Synset('gladden.v.01')]

'happiness' definition: state of well-being characterized by emotions ranging from contentment to intense joy
'joy' definition: the emotion of great happiness
```

```
# Run the Wu-Palmer similarity metric

print("Similarity between 'happiness' and 'joy' according to Wu-Palmer Similarity Metric:")
print(wn.wup_similarity(happiness_synset, joy_synset))
```

```
Similarity between 'happiness' and 'joy' according to Wu-Palmer Similarity Metric:
0.8
```

```
# Run the Lesk algorithm

from nltk.wsd import lesk
sent = ['She', 'felt', 'a', 'sense', 'of', 'happiness', 'at', 'the', 'party', '.']

result = lesk(sent, 'happiness', 'n')
print("Lesk algorithm result for 'happiness':")
print(str(result) + ': ' + str(result.definition()))
print()

sent[5] = 'joy'
result = lesk(sent, 'joy', 'n')
print("Lesk algorithm result for 'joy':")
print(str(result) + ': ' + str(result.definition()))
```

```
Lesk algorithm result for 'happiness':
Synset('happiness.n.02'): emotions experienced when in a state of well-being

Lesk algorithm result for 'joy':
Synset('joy.n.02'): something or someone that provides a source of happiness
```

The Wu-Palmer similarity metric seemed to be fairly accurate with these two words, as it said that 'joy' and 'happiness' were 80% similar. When analyzing these words with the Lesk algorithm, they were also fairly similar, with the output for 'joy' even containing the word 'happiness'. The main difference is that Lesk indicated that 'happiness' is an emotion, while 'joy' is something that causes the emotion.

9. Analysis with SentiWordNet

SentiWordNet is a resource that assigns sentiment scores to synsets. Sentiment scores indicate how likely it is that lemmas in this synset are being used positively, negatively, or objectively. SentiWordNet has many possible use cases in the real world today. For example, it would be useful in analyzing the emotion displayed in a tweet to determine how a user feels about a certain topic.

```
# Imports for SentiWordNet
nltk.download('sentiwordnet')
from nltk.corpus import sentiwordnet as swn
```

```
[nltk_data] Downloading package sentiwordnet to /root/nltk_data...
[nltk_data]   Package sentiwordnet is already up-to-date!
```

```
# Select an emotionally charged word and find its senti-synsets
senti_synsets = swn.senti_synsets('vicious')
print("Senti-synsets for the word 'vicious':")

# Output the polarity scores for each synset
for synset in senti_synsets:
  print()
  print(synset.synset)
  print("Positive score: ", synset.pos_score())
  print("Negative score: ", synset.neg_score())
  print("Objective score: ", synset.obj_score())
```

```
Senti-synsets for the word 'vicious':

Synset('barbarous.s.01')
Positive score:  0.0
Negative score:  0.625
Objective score:  0.375

Synset('evil.s.02')
Positive score:  0.0
Negative score:  0.5
Objective score:  0.5

Synset('condemnable.s.01')
Positive score:  0.0
Negative score:  0.875
Objective score:  0.125

Synset('poisonous.s.03')
Positive score:  0.0
Negative score:  0.75
Objective score:  0.25
```

```
# Make up a sentence
sentence = 'NLP is such an interesting and exciting course'

# Output the polarity for each word in the sentence
tokens = sentence.split()

neg = 0
pos = 0

for token in tokens:
  syn_list = list(swn.senti_synsets(token))
  if syn_list:
    syn = syn_list[0]
    print("Polarity scores for the word '" + token + "'")
    print("Positive score: ", syn.pos_score())
    print("Negative score: ", syn.neg_score())
    print("Objective score: ", syn.obj_score())
    print()
```

```
Polarity scores for the word 'NLP'
Positive score:  0.0
Negative score:  0.0
Objective score:  1.0

Polarity scores for the word 'is'
Positive score:  0.25
Negative score:  0.125
Objective score:  0.625

Polarity scores for the word 'such'
Positive score:  0.0
Negative score:  0.125
Objective score:  0.875

Polarity scores for the word 'an'
Positive score:  0.0
Negative score:  0.125
Objective score:  0.875

Polarity scores for the word 'interesting'
Positive score:  0.125
Negative score:  0.0
Objective score:  0.875

Polarity scores for the word 'exciting'
Positive score:  0.25
Negative score:  0.375
Objective score:  0.375

Polarity scores for the word 'course'
Positive score:  0.0
Negative score:  0.0
Objective score:  1.0
```

Overall, the output from SentiWordNet seemed accurate, but conservative. For example, SentiWordNet indicated that the word 'evil' is 50% likely to be negative and 50% likely to be objective. If I were to have chosen these numbers myself, I would have made it more likely to be negative, so it seems like SentiWordNet prefers to classify something as objective rather than making mistakes. From my sentence input, I observed that nouns are more likely to be scored as 100% objective than verbs and adjectives. I think that these scores would be useful in analyzing emotional texts, but they are ultimately limited by the conservative nature of the algorithm. For texts that are only slightly positive or slightly negative, the tool may not be that helpful in making a classification.

10. Collocations

Collocations are formed when two or more words occur together with a significant frequency, such as 'vice president'. It is important to know which words are part of a collocation, because you cannot logically substitute synonyms within them (e.g. 'break the ice' does not mean the same thing as 'shatter the ice'). For NLP applications like language translation, extracting collocations and processing them as one unit is essential and is often solved using frequency analysis.

```
# Necessary downloads for accessing text4 (the Inaugural corpus)
nltk.download('gutenberg')
nltk.download('genesis')
```

```
nltk.download('inaugural')
nltk.download('nps_chat')
nltk.download('webtext')
nltk.download('treebank')
nltk.download('stopwords')
```

```
[nltk_data] Downloading package gutenberg to /root/nltk_data...
[nltk_data]   Package gutenberg is already up-to-date!
[nltk_data] Downloading package genesis to /root/nltk_data...
[nltk_data]   Package genesis is already up-to-date!
[nltk_data] Downloading package inaugural to /root/nltk_data...
[nltk_data]   Package inaugural is already up-to-date!
[nltk_data] Downloading package nps_chat to /root/nltk_data...
[nltk_data]   Package nps_chat is already up-to-date!
[nltk_data] Downloading package webtext to /root/nltk_data...
[nltk_data]   Package webtext is already up-to-date!
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

```
# Output collocations for text4, the Inaugural corpus
from nltk.book import text4

text4.collocations()
```

```
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
```

```
# Select one of the collocations identified by NLTK
collocation = 'every citizen'

# Calculate mutual information using the formula log(P(x,y) / [P(x) * P(y)])
import math

text = ' '.join(text4.tokens)
vocab = len(set(text4))

# Calculate P(x,y)
Pxy = text.count(collocation) / vocab
print("p(every citizen) = ", Pxy)

# Calculate P(x)
Px = text.count(collocation.split()[0]) / vocab
print("p(every) = ", Px)

# Calculate P(y)
Py = text.count(collocation.split()[1]) / vocab
print('p(citizen) = ', Py)

print()

# Plug the above values into the final formula to get the result
mutual_info = math.log2(Pxy / (Px * Py))
print("Mutual information of 'every citizen' = ", mutual_info)
```

```
p(every citizen) =  0.0016957605985037406
p(every) =  0.03291770573566085
p(citizen) =  0.032618453865336655

Mutual information of 'every citizen' =  0.6593084177360863
```

Using the mutual information formula, we received a positive result of 0.6, indicating that these two words do form a collocation. However, this value is close to zero, so it is not a very strong determination. I would agree with this output, as 'every citizen' is not as common as a phrase like 'public debt', but I can understand why these words would frequently appear together in the Inaugural corpus.