## A WordNet Experience

WordNet is a Princeton project grouping English words in sets linked by semantics or lexical relationships. Words are often linked because they are synonyms into sets called synsets, which reveals a hierarchical structures in English. The database is useful, along with databases like nltk's text, to examine relationships between words. I am here to explore it!

### The Chosen Noun

For demonstration purposes, I'll select a noun to examine it's relationships, and it's synset. Lets start by printing as much information as we can about the noun, and traveling up the *hiearchy* that WordNet has found words form.

```
# Has nltk never been set up in this environment? Run these lines:
# (Here is a github issue about why omw-1.4 is seperate)
# https://github.com/nltk/nltk/issues/3024
import nltk
nltk.download("wordnet")
nltk.download("omw-1.4")

from nltk.corpus import wordnet as wn

# Now... what noun to use.... Asking ChatGPT for a random word, I got
back:
# "Sure, here's a random noun: "umbrella""
the_chosen_noun = "umbrella"

# Lets See it's synsets
synsets = wn.synsets(the_chosen_noun)
print(synsets)
```

```
[Synset('umbrella.n.01'), Synset('umbrella.n.02'),
Synset('umbrella.n.03'), Synset('umbrella.s.01')]
```

We can see that there are 4 different meanings for the word umbrella. I must admit I don't even know what the 's' means in the fourth meaning. Thats why we explore!

```
for synset in synsets:
  print(synset.name() + ": " + synset.definition())
```

```
umbrella.n.01: a lightweight handheld collapsible canopy
umbrella.n.02: a formation of military planes maintained over ground
operations or targets
umbrella.n.03: having the function of uniting a group of similar
things
umbrella.s.01: covering or applying simultaneously to a number of
similar items or elements or groups
```

Well, those are the definitions you would expect, but I still couldn't quite tell what 's' meant. I asked chatGPT again!

> In WordNet, the letter "s" in a synset name indicates that the synset represents an adjective satellite. An adjective satellite is a type of adjective that modifies the meaning of a noun by specifying a particular aspect or quality of it, rather than directly describing a property of the noun itself. Adjective satellites are often used to convey subtle shades of meaning or to express complex relationships between objects.

Considering we were tryinig to get down to the bottom of explaining what an adjective satellite was in class, this is quite a good example! If you had to think of a strange use case of umbrella, what would you think of? "Umbrella term" makes the noun *term* encompass a wide range of concepts or ideas. This particular example is fun, as umbrella is a noun in any other case!

For fun, lets compare umbrella's use as a noun and adjective satelite when it shares a similar meaning.

```
for synset in synsets:
  print(synset.examples())
```

```
[]
['an air umbrella over England']
['the Democratic Party is an umbrella for many liberal groups', 'under
the umbrella of capitalism']
['an umbrella organization', 'umbrella insurance coverage']
```

Look at that last example! You can tell that the noun organization has been modified to mean "An organization that covers simultaneously to a number of similar items" while no property of the noun was modified (We didn't say the size of the organization was large, we specified particular aspect that was before subtle). Compare that to the noun usage example above it, where umbrella was applied to the meaning of Democratic Party, but by simply saying it was an umbrella (noun).

All that nuance is fun, but lets pick the simple handheld collapsible canopy for our *chosen one* in the rest of this exercise.

```
the_chosen_noun = synsets[0]

def examine_synset(synset):
  # Lets print all we can about umbrellas!
  print("Definition: " + synset.definition())

  # Given that examples can be empty (and is), we should write a print
statement
  # that can handle any length of list
  examples = synset.examples()
  if len(examples):
    separator = ", "
```

```
    formatted_examples = separator.join(synset.examples())
    print("Examples: " + formatted_examples)
  else:
    print("Examples: N/A")

  # Same case for lemmas, we may not have lemmas
  lemmas = synset.lemmas()
  if len(lemmas):
    separator = ", "
    formatted_lemmas = separator.join([lemma.name() for lemma in
lemmas])
    print("Lemmas: " + formatted_lemmas)
  else:
    print("Examples: N/A")

examine_synset(the_chosen_noun)
```

```
Definition: a lightweight handheld collapsible canopy
Examples: N/A
Lemmas: umbrella
```

Knowing that about our noun, lets traverse the hiearchy by travelling up the 'hypernyms' of the word (or the list of synsets that represent more general concepts that the current sysnset is part of).

```
# If a word has multiple hypernyms, we will select the first hypernym
# Select a starting hypernym, assuming there is one (mimic a do-while
loop)
# then just keep printing the next hypernym until there are none left
def traverse(synset) :
  hyper = synset.hypernyms()[0]
  while hyper:
    print(hyper)
    if not hyper.hypernyms():
      break
    hyper = hyper.hypernyms()[0]

traverse(the_chosen_noun)

# Read that code aloud and quickly to sound very hyper
```

```
Synset('canopy.n.03')
Synset('shelter.n.02')
Synset('protective_covering.n.01')
Synset('covering.n.02')
Synset('artifact.n.01')
Synset('whole.n.02')
Synset('object.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

You can see here that there is a clear hiearchy stemming from 'umbrella'. Wordnet's nouns are densly connected in this way, where nouns at the top of the hiearchy are general nouns, with specificity growing as you go down. In that way you could call the noun wordnet a tree. Note that it is also doubly linked, with the ability to traverse in any direction on semantic relations. Those relations are:

- hypernyms: A list of synsets that represent more general concepts that the current synset is a type of.
- hyponyms: A list of synsets that represent more specific concepts that are types of the current synset.
- holonyms: A list of synsets that represent the whole that the current synset is a part of (e.g. 'tree' is a holonym of 'branch').
- meronyms: A list of synsets that represent the parts that make up the current synset (e.g. 'branch' is a meronym of 'tree').

I could also imagine traversing the net through a words lemmas, attributes, or antonyms.

For example, lets look at the relationships to our word!

```
# Prints a list attribute of a predefined synset (the_chosen_word)
def synset_list(attribute, synset):
  if not hasattr(synset, attribute):
    print(synset.name() + " has no attribute: " + attribute)
    return
  # Note that I both get the attribute  (a function) and run it with
'()'
  list = getattr(synset, attribute)()
  if not list:
    print("There are no " + attribute + " of the word " +
synset.name())
  else:
    separator = ", "
    print(attribute + ": " +
          separator.join([element.name() for element in list]))


# It's worth noting I was told to ouput an empty list if none exist
but this
# seems like a better way to communicate to the user if this function
were
# implemented elsewhere.
synset_list("hypernyms", the_chosen_noun)
synset_list("hyponyms", the_chosen_noun)
synset_list("meronyms", the_chosen_noun)
synset_list("holonyms", the_chosen_noun)
```

```
hypernyms: canopy.n.03
hyponyms: gamp.n.01
```

```
umbrella.n.01 has no attribute: meronyms
umbrella.n.01 has no attribute: holonyms
```

## The Chosen Verb!

Now we move on to verbs! Each part of speech in WordNet has it's own organization, so its worth looking at another section. Asking ChatGPT, our chosen verb is "Swim"! I'll go ahead use the lowercase "swim". Let's examine it as we did before:

```
the_chosen_verb = 'swim'

# All of the synsets
print(wn.synsets('swim'))

# Just the verbs!
synsets = wn.synsets('swim', pos=wn.VERB)
print(synsets)
```

```
[Synset('swimming.n.01'), Synset('swim.v.01'), Synset('float.v.02'),
Synset('swim.v.03'), Synset('swim.v.04'), Synset('swim.v.05')]
[Synset('swim.v.01'), Synset('float.v.02'), Synset('swim.v.03'),
Synset('swim.v.04'), Synset('swim.v.05')]
```

```
# Picking the first synset
the_chosen_verb = synsets[0]
# The functions from earlier!
examine_synset(the_chosen_verb)
traverse(the_chosen_verb)
```

```
Definition: travel through water
Examples: We had to swim for 20 minutes to reach the shore, a big fish
was swimming in the tank
Lemmas: swim
Synset('travel.v.01')
```

```
wn.synset('travel.v.01').hypernyms()
```

```
[]
```

I couldn't believe that there weren't any more hypernyms for the word swim so I had to check! It goes to show that while wordnet is organized similarly for verbs as it is for nouns, it is less dense so there isn't some *umbrella* noun that generalizes the meaning of every verb. In our case, there isn't a more general action the 'travel'. For an example of a verb hiearchy, think of the word 'whisper', which might have a hiearchy like:

```
['whisper', 'talk', 'communicate', 'interact']
```

However, while the tree is less tall, it is more horizontally linked, with more lemmas for a given word. A good example is how each verb could be linked by it's verb_frame, which represents the syntactic pattern a verb might be used in. I'd say noun relationships seem more reliable in my experience, all things considered.

### Morphy?

So, usually this tool can find the base form of a word using a set of rules of detachment. In my case, I chose the word 'swim' which is already the base form for the general action of swimming so I can't get any exciting results. I will note, that if I wanted to find every form, I could input the string into morphy, and keep calling morphy with a null string argument, which would keep giving base forms. *Or so I thought*

Looking through the NLTK github, it seems the ability to loop through words with multiple base forms is not implemented in the nltk implementation of wordnet! The princeton documentation states:

> The first time that Morphy is called with a specific string, it returns a base form. For each subsequent call to Morphy made with a NULL string argument, Morphy returns another base form

However, the code of nltk is:

```
if pos is None:
            morphy = self._morphy
            analyses = chain(a for p in POS_LIST for a in morphy(form,
p))
        else:
            analyses = self._morphy(form, pos, check_exceptions)

        # get the first one we find
        first = list(islice(analyses, 1))
        if len(first) == 1:
            return first[0]
        else:
            return None
```

Which gives no way to access the other analyzed forms! Just the first one returned. However, it reveals we can use a less friendly private function: _morphy. It doesn't add a case for not inputing pos, so we must enter pos. I'll exemplify it's use with axes, which is the example word from the princeton implementation

```
print(wn._morphy('axes', pos='n'))
print(wn._morphy('axes', pos='v'))
```

```
['ax', 'axis']
['axe', 'ax']
```

For the word swim, well...

```
print(wn._morphy('swim', pos='n'))
print(wn._morphy('swim', pos='v'))
print(wn._morphy('swim', pos='a'))
```

```
['swim']
['swim']
[]
```

We find that swim is already a base form (at least in english).

## Similarity and meaning

For an example in using wordnet to calculate the meaning of words given their relation to other words, we'll be running the Wu-Palmer similarity metric and Lesk algorithm. For my two words, I think I'll go ahead pick '*weather*' and '*climate*'.

```
# Checking we have the right synsets
weather = wn.synsets('weather', pos='n')[0]
print(weather.definition())
climate = wn.synsets('climate', pos='n')[0]
print(climate.definition())
```

```
the atmospheric conditions that comprise the state of the atmosphere
in terms of temperature and wind and clouds and precipitation
the weather in some location averaged over some long period of time
```

### Lesk

The Lesk algorithm for Word Sense Disambiguation (WSD) takes the context a sentance appears in and gives you the synset with the highest number of overlapping words between the context sentence and the definitions of each synset. Now I know the definition I want to find above, but using lesk lets see if we can find that sysnset with just a sentance.

As a human, I know that "the wonderful weather we are having" refers to the temperature and etc. But will Lesk see that with just definitions

```
from nltk.wsd import lesk
sent = ['Wonderful', 'weather', 'we', 'are', 'having', '.']
print(lesk(sent, 'weather', 'n'))
```

```
Synset('weather.n.01')
```

It's right! We were looking for the first definition of weather! What about "The climate of this place is too hot for me?" with the word climate

```
from nltk import word_tokenize
sent = "The climate of this place is too hot for me?"
token = word_tokenize(sent)
print(lesk(token, 'climate', 'n'))
```

```
Synset('climate.n.01')
```

Right again!

Honest I intentionally used context that didn't have too many overlapping words, so the success of the Lesk algorithm was nice and unexpected. Out of curiosity I looked at the code:

```
def lesk(context_sentence, ambiguous_word, pos=None, synsets=None):
context = set(context_sentence)
    if synsets is None:
        synsets = wordnet.synsets(ambiguous_word)

    if pos:
        synsets = [ss for ss in synsets if str(ss.pos()) == pos]

    if not synsets:
        return None

    _, sense = max(
        (len(context.intersection(ss.definition().split())), ss) for
ss in synsets
    )

    return sense
```

And I can see it simply just finds the synset definition with the maximum intersection length... shame! I imagine that since my definitions were the first in the synsets, the algorithm selects them with priority.

### Wu Palmer

This similarity algorithm compares how similar two word senses are based on the depth of two sens in the taxonomy and that of their least common subsumer (most specific ancestor node). By comparing how much the depth of both differs from their least common 'ancestor' we can see some form of similarity. Words that are more similar, will have LCS closer to both the word's depth. Lets see:

```
# Fun note from looking at source code, this function is also a
# class method of synset (synset.wup_similarity(...) works)
print(wn.wup_similarity(weather, climate))
```

```
0.13333333333333333
```

From that metric, I would have to say I thought they would have to be closer together. If we traverse the tree for each word we can see how distant their common ancestor is:

```
print("Weather: ")
traverse(weather)
print("Climate: ")
traverse(climate)
```

```
Weather:
Synset('atmospheric_phenomenon.n.01')
Synset('physical_phenomenon.n.01')
```

```
Synset('natural_phenomenon.n.01')
Synset('phenomenon.n.01')
Synset('process.n.06')
Synset('physical_entity.n.01')
Synset('entity.n.01')
Climate:
Synset('environmental_condition.n.01')
Synset('condition.n.01')
Synset('state.n.02')
Synset('attribute.n.02')
Synset('abstraction.n.06')
Synset('entity.n.01')
```

Wordnet's hypernym relationship doesn't seem to be the best at modeling this. NLTK has more algorithms like Lin Similarity and Jiang-Conrath Similarity that may be better for this kind of relation but it's outside the scope of this experiement.

## SentiWordNet

SentiWordNet is a functionality built on top of the word net database that adds a score for emotional sentiment. By knowing if a word is "positive", "negative", or "neutral" you can analyze the sentiment of sentences by the scores of each individual word. This could be applied to analyzing the sentiment of movie reviews or censoring text.

For an example, if I pick an emotionally charged word like 'mourn' I could examine it's sentiment using this library.

```
from nltk.corpus import sentiwordnet as swn
nltk.download('sentiwordnet')

# I like the second definition word so I am picking index 1
print([synset.definition() for synset in wn.synsets("mourn")])
mourn = wn.synsets("mourn")[1]
print(mourn.name() + ": " + mourn.definition())

sentiment = swn.senti_synset(mourn.name())
print(sentiment)
```

```
['feel sadness', 'observe the customs of mourning after the death of a
loved one']
mourn.v.02: observe the customs of mourning after the death of a loved
one
<mourn.v.02: PosScore=0.0 NegScore=0.0>

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

And look at that... observing the sustoms of mourning after the death of a loved one has no negative connotation... Guess I'll pick the first meaning.

```
mourn = wn.synsets("mourn")[0]
print(mourn.name() + ": " + mourn.definition())

sentiment = swn.senti_synset(mourn.name())
print(sentiment)
```

```
mourn.v.01: feel sadness
<mourn.v.01: PosScore=0.0 NegScore=0.75>
```

Overall, the word has a negative polarity. Lets analyze a sentance. I was instructed to output the polarity for each word in the sentence, so I do. but note that a lot of words just don't have sentiment, and then I also just select the most common (first) sentiment returned by `senti_synsets`.

```
from tkinter.constants import E
sent = "We return home to mourn the dead"
neg = 0
pos = 0
tokens = word_tokenize(sent)
print(tokens)
for token in tokens:
  syn_list = list(swn.senti_synsets(token))
  if syn_list:
    syn = syn_list[0]
    print("The polarity of\t", token, "is negative:", syn.neg_score(),
"and positive:", syn.pos_score())
  else:
    print("No polarity for\t", token)
```

```
['We', 'return', 'home', 'to', 'mourn', 'the', 'dead']
No polarity for  We
The polarity of  return is negative: 0.0 and positive: 0.0
The polarity of  home is negative: 0.0 and positive: 0.0
No polarity for  to
The polarity of  mourn is negative: 0.75 and positive: 0.0
No polarity for  the
The polarity of  dead is negative: 0.0 and positive: 0.0
```

A complex NLP algorithm would be able to interpret these scores much beyond their basic polarity. I can see an algorithim factoring in sentiment scores into word sense disambiguiation, by choosing a word sense given a found sentiment of the sentence. Or it could at scale be used to categorize the setiment of texts. Large enough corpora could actually reveal complex changes in sentiment over time. Given that sentiment is an aspect of language NLP can struggle with, this kind of encoding has many applications overall.

## Collocations

A collocation is when two or more words work together to form some unique meaning, where you can not simply replace one word with a synonym. Words that appear together frequently often take up their own meaning. "Heavy rain" is a collocation that could relate

to the meaning of the word "downpour" and has it's own frequent use in the English langauge.

Lets output collocations for text4, the inaugrural corpus within NLTK. We can then select one of the collocations and calculate mutual information.

```
# Ensure you have nltk.book installed and text4 imported
nltk.download("book")
from nltk.book import *
```

```
[nltk_data] Downloading collection 'book'
[nltk_data]    |
[nltk_data]    | Downloading package abc to /root/nltk_data...
[nltk_data]    |   Package abc is already up-to-date!
[nltk_data]    | Downloading package brown to /root/nltk_data...
[nltk_data]    |   Package brown is already up-to-date!
[nltk_data]    | Downloading package chat80 to /root/nltk_data...
[nltk_data]    |   Package chat80 is already up-to-date!
[nltk_data]    | Downloading package cmudict to /root/nltk_data...
[nltk_data]    |   Package cmudict is already up-to-date!
[nltk_data]    | Downloading package conll2000 to /root/nltk_data...
[nltk_data]    |   Package conll2000 is already up-to-date!
[nltk_data]    | Downloading package conll2002 to /root/nltk_data...
[nltk_data]    |   Package conll2002 is already up-to-date!
[nltk_data]    | Downloading package dependency_treebank to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package dependency_treebank 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 gutenberg to /root/nltk_data...
[nltk_data]    |   Package gutenberg is already up-to-date!
[nltk_data]    | Downloading package ieer to /root/nltk_data...
[nltk_data]    |   Package ieer 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 movie_reviews to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package movie_reviews 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 names to /root/nltk_data...
[nltk_data]    |   Package names is already up-to-date!
[nltk_data]    | Downloading package ppattach to /root/nltk_data...
[nltk_data]    |   Package ppattach is already up-to-date!
[nltk_data]    | Downloading package reuters to /root/nltk_data...
[nltk_data]    |   Package reuters is already up-to-date!
[nltk_data]    | Downloading package senseval to /root/nltk_data...
[nltk_data]    |   Package senseval is already up-to-date!
[nltk_data]    | Downloading package state_union to /root/nltk_data...
[nltk_data]    |   Package state_union is already up-to-date!
[nltk_data]    | Downloading package stopwords to /root/nltk_data...
```

```
[nltk_data]    |   Package stopwords is already up-to-date!
[nltk_data]    | Downloading package swadesh to /root/nltk_data...
[nltk_data]    |   Package swadesh is already up-to-date!
[nltk_data]    | Downloading package timit to /root/nltk_data...
[nltk_data]    |   Package timit 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 toolbox to /root/nltk_data...
[nltk_data]    |   Package toolbox is already up-to-date!
[nltk_data]    | Downloading package udhr to /root/nltk_data...
[nltk_data]    |   Package udhr is already up-to-date!
[nltk_data]    | Downloading package udhr2 to /root/nltk_data...
[nltk_data]    |   Package udhr2 is already up-to-date!
[nltk_data]    | Downloading package unicode_samples to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package unicode_samples 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 wordnet to /root/nltk_data...
[nltk_data]    |   Package wordnet is already up-to-date!
[nltk_data]    | Downloading package wordnet_ic to /root/nltk_data...
[nltk_data]    |   Package wordnet_ic is already up-to-date!
[nltk_data]    | Downloading package words to /root/nltk_data...
[nltk_data]    |   Package words is already up-to-date!
[nltk_data]    | Downloading package maxent_treebank_pos_tagger to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package maxent_treebank_pos_tagger is already up-
[nltk_data]    |       to-date!
[nltk_data]    | Downloading package maxent_ne_chunker to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package maxent_ne_chunker is already up-to-date!
[nltk_data]    | Downloading package universal_tagset to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package universal_tagset is already up-to-date!
[nltk_data]    | Downloading package punkt to /root/nltk_data...
[nltk_data]    |   Package punkt is already up-to-date!
[nltk_data]    | Downloading package book_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package book_grammars is already up-to-date!
[nltk_data]    | Downloading package city_database to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package city_database is already up-to-date!
[nltk_data]    | Downloading package tagsets to /root/nltk_data...
[nltk_data]    |   Package tagsets is already up-to-date!
[nltk_data]    | Downloading package panlex_swadesh to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package panlex_swadesh is already up-to-date!
[nltk_data]    | Downloading package averaged_perceptron_tagger to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Package averaged_perceptron_tagger is already up-
```

```
[nltk_data]    |       to-date!
[nltk_data]    |
[nltk_data]  Done downloading collection book
```

```
text4.collocation_list()
```

```
[('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')]
```

I feel like picking the "Almighty God" collocation for my analysis. Mutual information refers to the point-wise mutual information (pmi). PMI, found via the equation:

$$\log_2 \frac{P(x,y)}{P(x) \cdot P(y)}$$

This translates to the porbability of x and y occurring next to each other, divided by the probability of x multiplied by the probability of y. If the result is positive, then we know that x and y occur more together than expected by chance. And thus we have a collocation. Lets test on 'Almighty' and 'God'

```
from nltk.util import ngrams
import math

# The number of unigrams. In the text book this number was slightly
different
# but I don't see why so I am keeping just the length of the text.
unigrams = len(text4)

# The number of bigrams
bigrams = list(ngrams(text4, 2))
num_bigrams = len(bigrams)
```

```
# The probability of x or Almighty
px = text4.count("Almighty")/unigrams
# print(px)

# The probaibility of y or God
py = text4.count("God")/unigrams
# print(py)

# The probability of the bigram of xy or Almighty God
pxy = bigrams.count(("Almighty", "God"))/num_bigrams
# print(pxy)

# Calculated formula
pmi = math.log2(pxy/(px*py))
print("Our PMI is:", pmi)
```

```
Our PMI is: 9.527367559367379
```

The final calculation of the mutual information is rather high, suggesting that (at least in this text) "Almighty God" is a collocation. I'd say that this measurement is useful for just finding general sayings in English that aren't very well stated in the grammar and syntax. I know I only ever hear almighty in before God. So much to the point where the name of the film *Bruce Almighty* hints at it's plot involving god with just the name.

## Conclusion

These are all very good tools. Having a pre-annotated collection of English words that can be used for NLP is very helpful, especially with how easy it is to use the database inside of nltk. I'll have to keep the vocabulary and functions learned in this experiement close to my brain as I implement more and more NLP alorithms.