Module 2

Priya TV
Assistant Professor
Dept. of AIML
RVCE

Lexical semantics

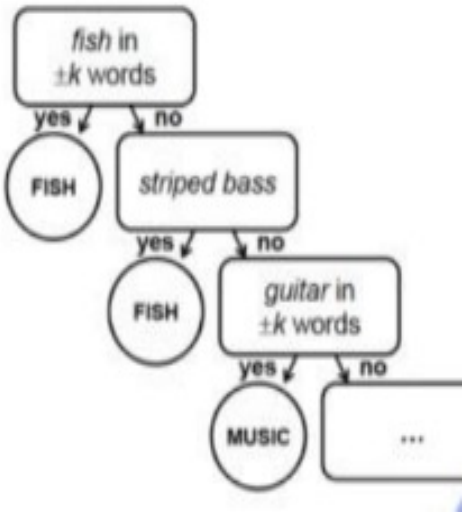
- Semantic meaning among lexical items (lexeme---individual entry)
- Lexeme will have form (how do I spell it) and meaning (sense).
- A lexeme with a different meaning is treated as a different lexeme
- Eg : Mouse, Bank

Definitions from the American Heritage Dictionary (Morris, 1985)

- **right** *adj.* located near the right hand esp. being on the right when facing the same direction as the observer
- **left** *adj.* located near to this side of the body than the right
- **red** *n.* the color of blood or a ruby
- **blood** *n.* the red liquid that circulates in the heart, arteries and veins of animals

Example: discriminating between bass (fish) and bass (music):

Context	Sense
fish in ±*k* words	FISH
striped bass	FISH
guitar in ±*k* words	MUSIC
bass player	MUSIC
piano in ±*k* words	MUSIC
sea bass	FISH
play bass	MUSIC
river in ±*k* words	FISH
on bass	MUSIC
bass are	FISH



Decision List algorithm

- Based on ‘one sense per collocation’ property
- Nearby words provide strong and consistent clues as to the sense of a target word
- Collect a large set of collocations for the ambiguous word
- Calculate word-sense probability distributions for all such collocations
- Calculate the log-likelihood ratio
- Log{P(sense_a/collocation_i)/P(sense_b/collocation_i)}
- Heigher log-likelihood implies more predictive evidence

Relation between word meanings

- Homonymy
- Polysemy
- Synonymy
- Antonymy
- Hyponymy
- Hypernymy
- Meronymy

Homonymy

- Defined as a relation that holds between words that have the same form with unrelated meanings
- Eg: bank (financial institution) or (riverside)
- bat (wooden stick-like thing) or (flying mammal thing)

Homophones --- write vs right or piece vs peace

Homographs ---bass (fish or guitar)

This can create problem for text to speech or information retrieval or speech recognition

Training for Naïve Bayes

- 'f' is a feature vector consisting of :

1. POS of W
2. Semantic and syntactic features of w
3. Collocation vector (set of words around it) and their POS
4. Co-occurrence vector

- Set parameters of Naïve Bayes using the maximum likelihood estimation (MLE) from the training data

$$P(s_i\ ) = count(s_i, w_j)/count(w_j)$$

$$P(f_i|s_i\ ) = count(f_i, s_i)/count(s_i)$$

Naïve Bayes for WSD

- A Naïve Bayes classifier chooses the most likely sense for a word given the features of the context.

$$\hat{s} = argmax\, P(s|f)$$

- Using Bayes' law, this can be expressed as :

$$\hat{s} = argmax\, P(s) \prod_{j=1}^{n} P(f_t|s)$$

- The Naïve' assumption: all the features are conditionally independent, given the sense:

Polysemy

- Two words are in same form and slightly different meanings
- Eg: The bank was constructed in 1987 out of local red brick
  I withdrew money from the bank

Sense 1: building belonging to the financial institution

Sense 2---a financial institution

Zeugma test

- Which of these flights *serve* breakfast?
- Does Midwest Express *serve* Philadelphia?

Synonymy

- Two words have very similar meaning but different forms
- Couch /sofa
- Big/large
- Automobile/car
- Water/h2o
- But cant be used in all context (eg h2o in scientific context else water)

	Sense1---finance	Sense2---river
Money	+1	0
Interest	+1	0
Fetch	0	0
Annual	+1	0
total	3	0

Walker's algorithm

- A thesaurus based approach
  step 1--- for each sense of the target word find the thesaurus category to which that sense belongs
- Step 2---calculate the score for each sense by using the context words. A context word will add 1 to the score of the sense if the thesaurus category of the word matches that of the sense.
- Eg: the money in this bank fetches an interest of 8% per annum
- Target word---bank
- Clue words---money, interest, annum,fetch

Consider the words *big* and *large*.

Are they synonyms?

- How **big** is that plane?
- Would I be flying on a **large** or small plane?

How about here?

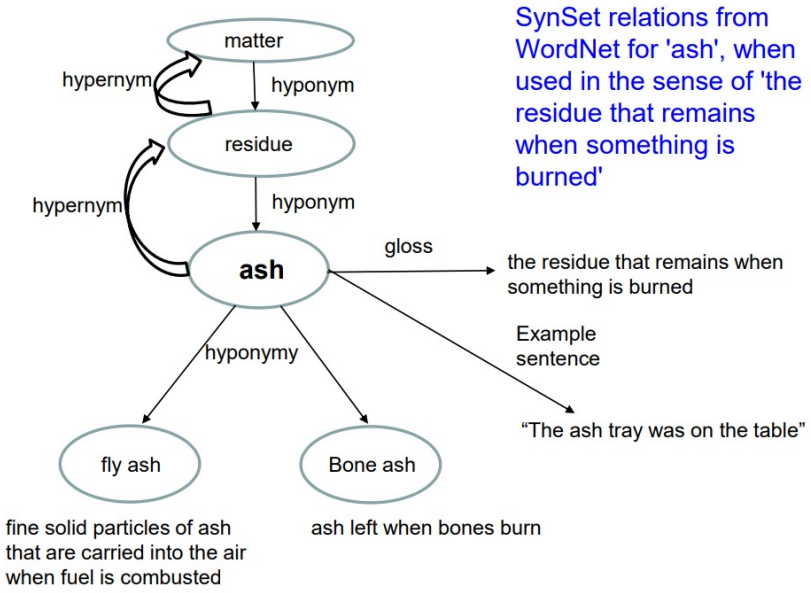
- Miss Nelson, for instance, became a kind of **big** sister to Benjamin.
- *Miss Nelson, for instance, became a kind of **large** sister to Benjamin.

Antonyms

- Senses that are opposite with respect to one feature of their meaning
- Otherwise, they are similar
- Dark/light
- Short/long
- Hot/cold
- In/out
- Up/down

Eg---hot/cold are related (they are temperature but 2 extremes)

Wordnet subgraph



From Wordnet

- The noun ash has 3 senses (first 2 from tagged texts)
- 1. (2) ash -- (the residue that remains when something is burned)
- 2. (1) ash, ash tree -- (any of various deciduous pinnate-leaved ornamental or timber trees of the genus Fraxinus)
- 3. ash -- (strong elastic wood of any of various ash trees; used for furniture and tool handles and sporting goods such as baseball bats)
- The verb ash has 1 sense (no senses from tagged texts)
- 1. ash -- (convert into ashes)

Hyponymy and hypernymy

- Hyponymy---subclass of other
- Eg: car is a hyponymy of vehicle, dog is a hyponymy of animal

- Hypernymy---vehicle is a hypernym of car, animal is a hypernym of dog

Meronyms and holonyms

- Transitive relation between senses
- X is a meronym of Y if it denotes a part of Y
- Reverse is holonyms

meronym	holonym
porch	house
wheel	car
leg	chair
nose	face

Lesk's algorithm

- Sense bag---contains the words in the definition of the candidate sense of the ambiguous words
- Context bag---contains the words in the definition of each sense of each context word
- Eg : on burning **coal** we get ash

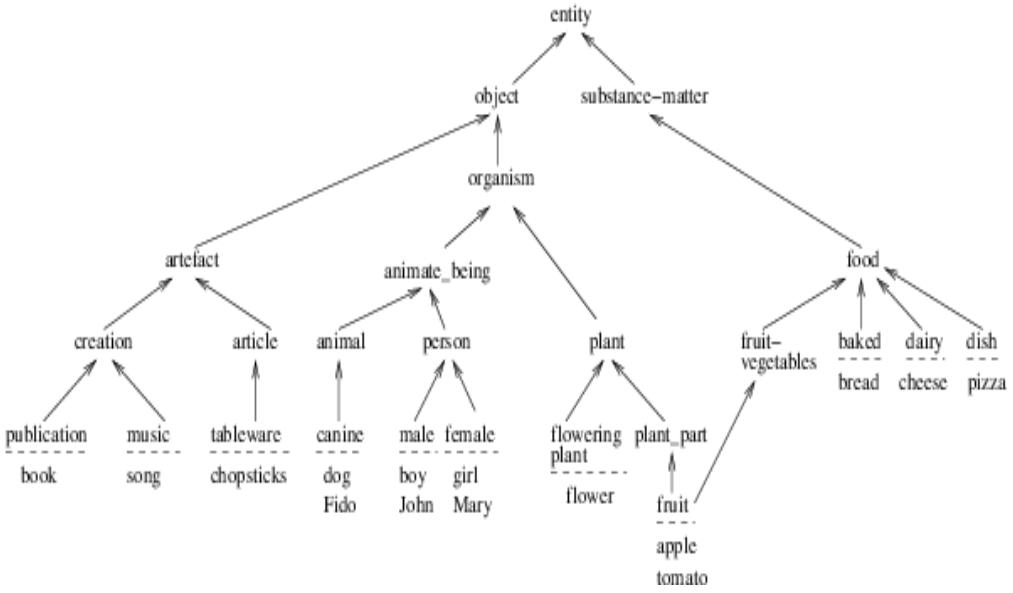
Knowledge-based approaches

- 1. Overlap based approach
- Require a machine readable dictionary
- Find the overlap between the features of different senses of an ambiguous word (sense bag) and the features of the word in its context (context bag)
- The features could be sense definitions , example sentences, hypernyms etc.
- The features could also be given weights
- The sense which has the maximum overlap is selected as the contextually appropriate sense

Wordnet

https://wordnet.Princeton.edu/wordnet/

- A hierarchically organized lexical database
- A machine readable thesaurus, dictionary
- Version of other languages are under development
- Synset---a set of synonyms representing a sense



Algorithms

- Knowledge-based approaches
- Machine learning based approaches

1. Supervised approaches
2. Semi-supervised algorithms]
3. Unsupervised algorithms

- Hybrid approaches

Word sense disambiguation

- Many words have several meaning or senses
- Eg: bass (guitar) and bass(fish)
- Disambiguation---task is to determine which of the senses of an ambiguous word is invoked in a particular use of the word
- This is done by looking at the context of the word's use.

Tokenizing text and wordnet basics

- Natural Language ToolKit (NLTK)---python library for NLP and text analytics.
- Tokenization is a method of breaking up a piece of text into many pieces, such as sentences and words.
- WordNet is a dictionary designed for programmatic access by natural language processing systems. It has many different use cases, including:

  - Looking up the definition of a word
  - Finding synonyms and antonyms
  - Exploring word relations and similarity
  - Word sense disambiguation for words that have multiple uses and definitions

Tokenizing text into sentences

- Corpus---body of text.
- Example:

```
para = "Hello World. It's good to see you. Thanks for buying this book."
```

To split into sentences, do the following code:

```
>>>from nltk.tokenize import sent_tokenize
>>> sent_tokenize(para)
['Hello World.', "It's good to see you.", 'Thanks for buying this book.']
```

```
from nltk.corpus import wordnet

class AntonymReplacer(object):
  def replace(self, word, pos=None):
    antonyms = set()
    for syn in wordnet.synsets(word, pos=pos):
      for lemma in syn.lemmas():
        for antonym in lemma.antonyms():
          antonyms.add(antonym.name())
    if len(antonyms) == 1:
      return antonyms.pop()
    else:
      return None

  def replace_negations(self, sent):
    i, l = 0, len(sent)
    words = []
    while i < l:
      word = sent[i]
      if word == 'not' and i+1 < l:
        ant = self.replace(sent[i+1])
        if ant:
          words.append(ant)
          i += 2
          continue
      words.append(word)
      i += 1
    return words
```

Replacing negations with antonyms

```
>>> from replacers import AntonymReplacer
>>> replacer = AntonymReplacer()
>>> replacer.replace('good')
>>> replacer.replace('uglify')

'beautify'

>>> sent = ["let's", 'not', 'uglify', 'our', 'code']
>>> replacer.replace_negations(sent)

["let's", 'beautify', 'our', 'code']
```

The previous code is equivalent to this code (both gives the same result).

```
>>> import nltk.data
>>> tokenizer =
nltk.data.load('tokenizers/punkt/PY3/english.pickle')
>>> tokenizer.tokenize(para)
['Hello World.', "It's good to see you.", 'Thanks for buying this
book.']
```

- *Note: The `sent_tokenize` function uses an instance of `PunktSentenceTokenizer` from the `nltk.tokenize.punkt` module. This instance has already been trained and works well for many European languages. So it knows what punctuation and characters mark the end of a sentence and the beginning of a new sentence.*

Tokenizing sentences in other languages

```
>>> spanish_tokenizer = nltk.data.load('tokenizers/punkt/PY3/spanish.
pickle')
>>> spanish_tokenizer.tokenize('Hola amigo. Estoy bien.’)
```

```
['Hola amigo.', 'Estoy bien.’]
```

Note: You can see a list of all the available language tokenizers in `/usr/share/nltk_data/tokenizers/punkt/PY3` (or `C:\nltk_data\tokenizers\punkt\PY3`).

Replacing synonyms

```
class WordReplacer(object):
      def __init__(self, word_map):
            self.word_map = word_map
      def replace(self, word):
            return self.word_map.get(word, word)
>>> from replacers import WordReplacer
>>> replacer = WordReplacer({'bday': 'birthday'})
>>> replacer.replace('bday')
'birthday'
>>> replacer.replace('happy')
'happy'
```

```
import enchant
from nltk.metrics import edit_distance

class SpellingReplacer(object):
  def __init__(self, dict_name='en', max_dist=2):
    self.spell_dict = enchant.Dict(dict_name)
    self.max_dist = max_dist

  def replace(self, word):
    if self.spell_dict.check(word):
      return word
    suggestions = self.spell_dict.suggest(word)

    if suggestions and edit_distance(word, suggestions[0]) <=
      self.max_dist:
      return suggestions[0]
    else:
      return word
```

Tokenizing sentences into words

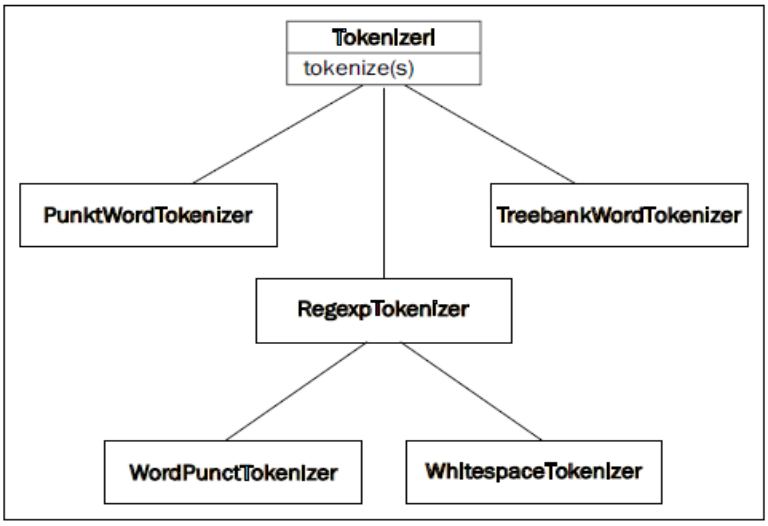
```
>>> from nltk.tokenize import word_tokenize
>>> word_tokenize('Hello World.')
['Hello', 'World', '.']
```

This is equivalent to the below code.

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize('Hello World.')
['Hello', 'World', '.']
```

Note: The `word_tokenize()` function is a wrapper function that calls `tokenize()` on an instance of the `TreebankWordTokenizer` class.

Inheritance tree for Tokenizer



Spelling correction with Enchant

- Enchant---a spelling correction API

```
>>> from replacers import SpellingReplacer
>>> replacer = SpellingReplacer()
>>> replacer.replace('cookbok')
'cookbook'
```

Removing repeating characters

```
>>> from replacers import RepeatReplacer
>>> replacer = RepeatReplacer()
>>> replacer.replace('looooove')
'love'
>>> replacer.replace('oooooh')
'oh'
>>> replacer.replace('goose')
'gose'
```

PunktWordTokenizer

- An alternative word tokenizer is `PunktWordTokenizer.`
- It splits on punctuation but keeps it with the word instead of creating separate tokens.

Example,

```
>>> from nltk.tokenize import PunktWordTokenizer
>>> tokenizer = PunktWordTokenizer()
>>> tokenizer.tokenize("Can't is a contraction.")
['Can', "'t", 'is', 'a', 'contraction.']
```

WordPunctTokenizer

- Another alternative word tokenizer is WordPunctTokenizer. It splits all punctuation into separate tokens:
- Example,

```
>>> from nltk.tokenize import WordPunctTokenizer
>>> tokenizer = WordPunctTokenizer()
>>> tokenizer.tokenize("Can't is a contraction.")
['Can', "'", 't', 'is', 'a', 'contraction', '.']
```

Replacement before tokenization

```
>>> from nltk.tokenize import word_tokenize
>>> from replacers import RegexpReplacer
>>> replacer = RegexpReplacer()
>>> word_tokenize("can't is a contraction")

['ca', "n't", 'is', 'a', 'contraction’]

>>> word_tokenize(replacer.replace("can't is a contraction"))

['can', 'not', 'is', 'a', 'contraction']
```

- Here is a simple usage example:

```
>>> from replacers import RegexpReplacer
>>> replacer = RegexpReplacer()
>>> replacer.replace("can't is a contraction")

'cannot is a contraction’

>>> replacer.replace("I should've done that thing I didn't do")

'I should have done that thing I did not do'
```

Tokenizing sentences using regular expressions

Regular expression---

- Example,

```
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer = RegexpTokenizer("[\w']+")
>>> tokenizer.tokenize("Can't is a contraction.")
["Can't", 'is', 'a', 'contraction’]
```

The above code is equivalent to the following:

```
>>> from nltk.tokenize import regexp_tokenize
>>> regexp_tokenize("Can't is a contraction.", "[\w']+")
["Can't", 'is', 'a', 'contraction']
```

Replacing words matching regular expressions

```
import re

replacement_patterns = [
  (r'won\'t', 'will not'),
  (r'can\'t', 'cannot'),
  (r'i\'m', 'i am'),
  (r'ain\'t', 'is not'),
  (r'(\w+)\'ll', '\g<1> will'),
  (r'(\w+)n\'t', '\g<1> not'),
  (r'(\w+)\'ve', '\g<1> have'),
  (r'(\w+)\'s', '\g<1> is'),
  (r'(\w+)\'re', '\g<1> are'),
  (r'(\w+)\'d', '\g<1> would')
]

class RegexpReplacer(object):
  def __init__(self, patterns=replacement_patterns):
    self.patterns = [(re.compile(regex), repl) for (regex, repl) in
      patterns]

  def replace(self, text):
    s = text
    for (pattern, repl) in self.patterns:
      s = re.sub(pattern, repl, s)
    return s
```

Stemming vs lemmatization

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('believes')
'believ'
>>> lemmatizer.lemmatize('believes')
'belief'
```

Training a sentence tokenizer

For this example, we'll be using the `webtext` corpus, specifically the `overheard.txt` file, so make sure you've downloaded this corpus. The text in this file is formatted as dialog that looks like this:

```
White guy: So, do you have any plans for this evening?
Asian girl: Yeah, being angry!
White guy: Oh, that sounds good.
>>> from nltk.tokenize import PunktSentenceTokenizer
>>> from nltk.corpus import webtext
>>> text = webtext.raw('overheard.txt')
>>> sent_tokenizer = PunktSentenceTokenizer(text)
>>> sents1 = sent_tokenizer.tokenize(text)
>>> sents1[0]
'White guy: So, do you have any plans for this evening?'
```

```
>>> from nltk.tokenize import sent_tokenize
>>> sents2 = sent_tokenize(text)
>>> sents2[0]
• 'White guy: So, do you have any plans for this evening?'
>>> sents1[678]
'Girl: But you already have a Big Mac...'
>>> sents2[678]
'Girl: But you already have a Big Mac...\\nHobo: Oh, this is all
theatrical.'
```

Lemmatizing words with WordNet

- A lemma is a root word, as opposed to the root stem. So unlike stemming, you are always left with a valid word that means the same thing.

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize('cooking')
'cooking'
>>> lemmatizer.lemmatize('cooking', pos='v')
'cook'
>>> lemmatizer.lemmatize('cookbooks')
'cookbook'
```

The SnowballStemmer class

- The `SnowballStemmer` class supports 13 non-English languages. It also provides two English stemmers: the original porter algorithm as well as the new English stemming algorithm.

```
>>> from nltk.stem import SnowballStemmer
>>> SnowballStemmer.languages('danish', 'dutch', 'english', 'finnish',
'french', 'german', 'hungarian', 'italian', 'norwegian', 'porter',
'portuguese', 'romanian', 'russian', 'spanish', 'swedish')
>>> spanish_stemmer = SnowballStemmer('spanish')
>>> spanish_stemmer.stem('hola')
u'hol'
```

Filtering stopwords in a tokenized sentence

- Stopwords are common words that generally do not contribute to the meaning of a sentence.

- Example,

```
>>> from nltk.corpus import stopwords
>>> english_stops = set(stopwords.words('english'))
>>> words = ["Can't", 'is', 'a', 'contraction']
>>> [word for word in words if word not in english_stops]

["Can't", 'contraction']
```

```
>>> stopwords.fileids()
['danish', 'dutch', 'english', 'finnish', 'french',
'german','hungarian', 'italian', 'norwegian', 'portuguese',
'russian','spanish', 'swedish', 'turkish']
```

```
>>> stopwords.words('dutch')
['de', 'en', 'van', 'ik', 'te', 'dat', 'die', 'in', 'een',
'hij','het', 'niet', 'zijn', 'is', 'was', 'op', 'aan', 'met',
'als', 'voor','had', 'er', 'maar', 'om', 'hem', 'dan', 'zou', 'of',
'wat', 'mijn','men', 'dit', 'zo', 'door', 'over', 'ze', 'zich',
'bij', 'ook', 'tot','je', 'mij', 'uit', 'der', 'daar', 'haar',
'naar', 'heb', 'hoe','heeft', 'hebben', 'deze', 'u', 'want', 'nog',
'zal', 'me', 'zij','nu', 'ge', 'geen', 'omdat', 'iets', 'worden',
'toch', 'al', 'waren','veel', 'meer', 'doen', 'toen', 'moet',
'ben', 'zonder', 'kan','hun', 'dus', 'alles', 'onder', 'ja',
'eens', 'hier', 'wie', 'werd','altijd', 'doch', 'wordt', 'wezen',
'kunnen', 'ons', 'zelf', 'tegen','na', 'reeds', 'wil', 'kon',
'niets', 'uw', 'iemand', 'geweest','andere']
```

The RegexStemmer class

- It takes a single regular expression (either compiled or as a string) and removes any prefix or suffix that matches the expression:

```
>>> from nltk.stem import RegexpStemmer
>>> stemmer = RegexpStemmer('ing')
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookery'
>>> stemmer.stem('ingleside')
'leside'
```

Note: A `RegexpStemmer` class should only be used in very specific cases that are not covered by the `PorterStemmer` or the `LancasterStemmer` class because it can only handle very specific patterns and is not a general-purpose algorithm.

The LancasterStemmer class

- More aggressive thab PorterStemmer functions.

```
>>> from nltk.stem import LancasterStemmer
>>> stemmer = LancasterStemmer()
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookery'
```

Looking up Synsets for a word in WordNet

- WordNet is a lexical database for the English language. In other words, it's a dictionary designed specifically for natural language processing.
- NLTK comes with a simple interface to look up words in WordNet. What you get is a list of Synset instances, which are groupings of synonymous words that express the same concept.

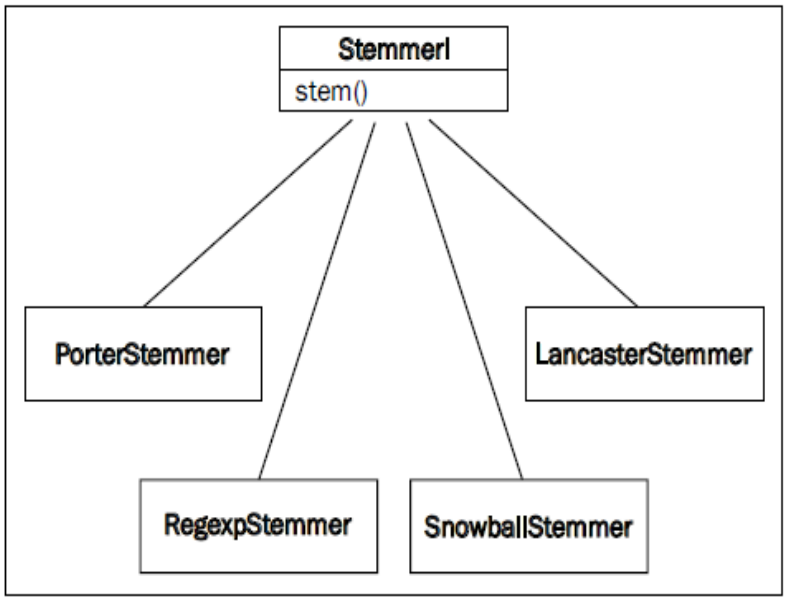
Example,

```
>>> from nltk.corpus import wordnet
>>> syn = wordnet.synsets('cookbook')[0]
>>> syn.name()

'cookbook.n.01'

>>> syn.definition()
'a book of recipes and cooking directions'
```

Working with hypernyms

- Hypernyms---most abstract terms
- Hyponyms--- more specific terms
- Hypernyms provide a way to categorize and group words based on their similarity to each other.

```
>>> syn.hypernyms()
[Synset('reference_book.n.01')]
>>> syn.hypernyms()[0].hyponyms()
[Synset('annual.n.02'), Synset('atlas.n.02'),
Synset('cookbook.n.01'), Synset('directory.n.01'),
Synset('encyclopedia.n.01'), Synset('handbook.n.01'),
Synset('instruction_book.n.01'), Synset('source_book.n.01'),
Synset('wordbook.n.01')]
>>> syn.root_hypernyms()
[Synset('entity.n.01')]
```



Stemming words

- Stemming is a technique to remove affixes from a word, ending up with the stem. For example, the stem of `cooking` is `cook`.
- One of the most common stemming algorithms is the Porter stemming algorithm by Martin Porter.

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookeri'
```

Note: There are other stemming algorithms out there besides the Porter stemming algorithm, such as the Lancaster stemming algorithm, developed at Lancaster University.

Part of speech (POS)

Part of speech	Tag
Noun	n
Adjective	a
Adverb	r
Verb	v

```
>>> len(wordnet.synsets('great'))
7
>>> len(wordnet.synsets('great', pos='n'))
1
>>> len(wordnet.synsets('great', pos='a'))
6
```

Looking up lemmas and synonyms in WordNet

```
>>> from nltk.corpus import wordnet
>>> syn = wordnet.synsets('cookbook')[0]
>>> lemmas = syn.lemmas()
>>> len(lemmas)
2
>>> lemmas[0].name()
'cookbook'
>>> lemmas[1].name()
'cookery_book'
>>> lemmas[0].synset() == lemmas[1].synset()
True
```

Replacing and correcting words

Discovering word collocations

- Collocations are two or more words that tend to appear frequently together, such as United States.
- i.e., common phrases that occur frequently throughout the text.

All possible synonyms

```
>>> synonyms = []

>>> for syn in wordnet.synsets('book'):

... for lemma in syn.lemmas():

... synonyms.append(lemma.name())

>>> len(synonyms)

38
```

Antonyms

```
>>> gn2 = wordnet.synset('good.n.02')
>>> gn2.definition()
'moral excellence or admirableness'
>>> evil = gn2.lemmas()[0].antonyms()[0]
>>> evil.name
'evil'
>>> evil.synset().definition()
'the quality of being morally wrong in principle or practice'
>>> ga1 = wordnet.synset('good.a.01')
>>> ga1.definition()
'having desirable or positive qualities especially those suitable for
a thing specified'
>>> bad = ga1.lemmas()[0].antonyms()[0]
>>> bad.name()
'bad'
>>> bad.synset().definition()
'having undesirable or negative qualities'
```

Calculating WordNet synset similarity

```
>>> from nltk.corpus import wordnet
>>> cb = wordnet.synset('cookbook.n.01')
>>> ib = wordnet.synset('instruction_book.n.01')
>>> cb.wup_similarity(ib)
0.9166666666666666
>>> dog = wordnet.synsets('dog')[0]
>>> dog.wup_similarity(cb)
0.38095238095238093
```

Note: dog and cookbook are apparently 38% similar because, they share common hypernyms.

```
>>> sorted(dog.common_hypernyms(cb))
[Synset('entity.n.01'), Synset('object.n.01'),
Synset('physical_entity.n.01'), Synset('whole.n.02')]
```