# Lexical Chains

Ana Maria Martinez Sidera
A20410762
Department of Computer Science
Illinois Institute of Technology

May 5, 2018

## 1. Create lexical chains

In this assignment we are going to create lexical chains based on the lexical relationships of synonymy, antonymy, and one level of hyper/hyponymy. First of all we have to go through the text taking only the words that are in Noun's position. For this we will have to take all the tags that wordnet gives us: NN (Noun, singular or mass), NNS (Noun, plural), NNP (Proper noun, singular), NNPS (Proper noun, plural).

```
position = ['NN', 'NNS', 'NNP', 'NNPS']

sentence = nltk.sent_tokenize(input_txt)
tokenizer = RegexpTokenizer(r'\w+')
tokens = [tokenizer.tokenize(w) for w in sentence]
tagged =[pos_tag(tok) for tok in tokens]
nouns = [word.lower() for i in range(len(tagged))
        for word, pos in tagged[i] if pos in position ]
```

After compute this script, we have the list of words that we have taken from the text.

```
['today', 'victory', 'party', 'celebration', 'freedom', 'end', 'beginning', 'renewal', 'change',
'i', 'solemn', 'forbears', 'century', 'world', 'man', 'power', 'forms', 'poverty', 'forms', 'life',
'beliefs', 'forebears', 'issue', 'globe', 'belief', 'rights', 'man', 'generosity', 'state', 'hand',
'god', 'today', 'heirs', 'revolution', 'word', 'time', 'place', 'torch', 'generation', 'americans',
'century', 'war', 'peace', 'proud', 'heritage', 'undoing', 'rights', 'nation', 'today', 'home',
'world', 'let', 'nation', 'price', 'meet', 'support', 'friend', 'foe', 'survival', 'success',
'liberty', 'allies', 'origins', 'share', 'loyalty', 'friends', 'host', 'ventures', 'challenge',
'odds', 'split', 'asunder', 'states', 'ranks', 'word', 'form', 'colonial', 'control', 'tyranny',
'view', 'freedom', 'power', 'back', 'tiger', 'people', 'huts', 'villages', 'globe', 'bonds', 'mass',
'misery', 'efforts', 'period', 'communists', 'votes', 'society', 'sister', 'republics', 'south',
'border', 'pledge', 'words', 'deeds', 'alliance', 'progress', 'men', 'governments', 'chains',
'poverty', 'revolution', 'hope', 'prey', 'powers', 'neighbors', 'aggression', 'subversion',
'americas', 'power', 'master', 'house', 'world', 'assembly', 'states', 'nations', 'hope', 'age',
'instruments', 'war', 'instruments', 'peace', 'pledge', 'support', 'forum', 'shield', 'area',
'writ', 'nations', 'pledge', 'request', 'sides', 'peace', 'powers', 'destruction', 'science',
'engulf', 'humanity', 'weakness', 'arms', 'doubt', 'doubt', 'groups', 'nations', 'comfort',
'course', 'sides', 'cost', 'weapons', 'spread', 'atom', 'balance', 'terror', 'hand', 'mankind',
'war', 'sides', 'civility', 'sign', 'weakness', 'sincerity', 'fear']
```

Figure 1: List of words.

Once we have all the names we will create lists of relationships. In these lists we will put all the synonyms, antonyms, hyponyms, hypernym of each of the words.

```
def relation_list(nouns):

```

```
    relation_list = defaultdict(list)

    for k in range (len(nouns)):
        relation = []
        for syn in wordnet.synsets(nouns[k], pos = wordnet.NOUN):
            for l in syn.lemmas():
                relation.append(l.name())
                if l.antonyms():
                    relation.append(l.antonyms()[0].name())
            for l in syn.hyponyms():
                if l.hyponyms():
                    relation.append(l.hyponyms()[0].name().split('.')[0])
            for l in syn.hypernyms():
                if l.hypernyms():
                    relation.append(l.hypernyms()[0].name().split('.')[0])
        relation_list[nouns[k]].append(relation)
    return relation_list
```

In the next Figure you can see an example of a relationship of the word victory. The script has gone word by word and we have looked at synonyms, antonyms, hypernyms and hyponyms.

```
Word: Victory
Synonyms:
   victory
   triumph
Antonyms:
   defeat
Hyponyms:
   takedown
   grand_slam
   first-place_finish
Hypernyms:
   happening
   happening
```

Figure 2: Relation for the word: victory.

To perform the lexical chain we have to compare three things for each word:

- All the nouns we have already in the chain. If there is a noun in the chain, we will have to add one more to the final count.
- If in the relations list of this noun, we find one word in the actual chain. We will append our noun in the same chain. Moreover, we compute a similarity for the two words. We have a threshold for determine if the two words can be in the same chain or not.
- If the nouns that we have already defined in each chain, we look in their relations and if the current noun is in that relation list we will append the noun to the same chain. Like in the other case, we compute a similarity between the two words.

In case that none of these relationships are not met we will put a new chain to our final vector.

```python
def create_lexical_chain(nouns, relation_list):
    lexical = []
    threshold = 0.5
    for noun in nouns:
        flag = 0
        for j in range(len(lexical)):
            if flag == 0:
                for key in list(lexical[j]):
                    if key == noun and flag == 0:
                        lexical[j][noun] +=1
                        flag = 1
                    elif key in relation_list[noun][0] and flag == 0:
                        syns1 = wordnet.synsets(key, pos = wordnet.NOUN)
                        syns2 = wordnet.synsets(noun, pos = wordnet.NOUN)
                        if syns1[0].wup_similarity(syns2[0]) >= threshold:
                            lexical[j][noun] = 1
                            flag = 1
                    elif noun in relation_list[key][0] and flag == 0:
                        syns1 = wordnet.synsets(key, pos = wordnet.NOUN)
                        syns2 = wordnet.synsets(noun, pos = wordnet.NOUN)
                        if syns1[0].wup_similarity(syns2[0]) >= threshold:
                            lexical[j][noun] = 1
                            flag = 1
        if flag == 0:
            dic_nuevo = {}
            dic_nuevo[noun] = 1
            lexical.append(dic_nuevo)
            flag = 1
    return lexical
```

```
Chain 1 : {'today': 1, 'time': 1, 'period': 1}
Chain 2 : {'victory': 1}
Chain 3 : {'party': 1}
Chain 4 : {'celebration': 1}
Chain 5 : {'freedom': 1, 'liberty': 1}
Chain 6 : {'end': 1, 'beginning': 1, 'change': 1, 'undoing': 1, 'origins': 1, 'efforts': 1, 'progress': 1,
'destruction': 1}
Chain 7 : {'renewal': 1}
Chain 8 : {'i': 1}
Chain 9 : {'solemn': 1}
Chain 10 : {'forbears': 1}
Chain 11 : {'century': 2}
Chain 12 : {'world': 1, 'man': 2, 'globe': 2, 'people': 1, 'mass': 1, 'men': 1, 'humanity': 1, 'mankind': 1}
Chain 13 : {'power': 1, 'powers': 2}
Chain 14 : {'forms': 2, 'form': 1, 'course': 1}
Chain 15 : {'poverty': 2}
```

Figure 3: Lexical chain.

We are going to prune our lexical chain. Since we want only the most important chains, we are going to delete from the list the chains that only have one element and that word is only repeated once.

```
def prune(lexical):
    final_chain = []
    while lexical:
        result = lexical.pop()
        if len(result.keys()) == 1:
            for value in result.values():
                if value != 1:
                    final_chain.append(result)
        else:
            final_chain.append(result)
    return final_chain
```

## 2. Results

In this section we are going to evaluate the results and comment on how they might be improved. We obtain the lexical chain with a depth level, we have several relationships such as the following case. Analyzing how the following chain has been created we have arrived at the following graph. The result obtained is quite good but it is quite far from

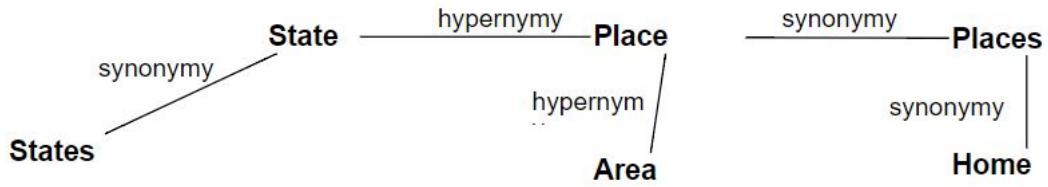


Figure 4: Lexical chain 6.

reality.

The depth levels should be much more extensive and not based only on the first step. We should create a tree and analyze it every time we have a new noun. It would also be good to analyze the similarity between each word. Each word in natural language is used in a different context and may even be synonymous, in the day to day we do not use it in the same way. We should take into account these small details to make a good lexical chain.

## 3. Extra credit

In this extra section we are going to use the lexical chains to automatically create a summary of the input article. The target of the automatic text summarizing is to reduce a

textual document to a summary that retains the important points of the original document. The algorithm that we are going to see tries to extract one or more sentences that cover the main topics of the original document using the idea that, if a sentences contains the most recurrent words in the text, it probably covers most of the topics of the text.

To make the summary we are going to use the lexical chain that we have created before. We will compute the frequency of all the words in the text, taking into account that if a word is in a chain we will count the sum of the whole chain. Once all the frequencies have been computed, we will normalize the values and make a filtering with a threshold for maximum and a threshold for minimum.

```
def return_frequencies(self, words, lexical_chain):
    frequencies = defaultdict(int)
    for word in words:
        for w in word:
            if w not in self._stopwords:
                flag = 0
                for i in lexical_chain:
                    if w in list(i.keys()):
                        frequencies[w] = sum(list(i.values()))
                        flag = 1
                        break
                if flag == 0:
                    frequencies[w] += 1
    m = float(max(frequencies.values()))
    for w in list(frequencies.keys()):
        frequencies[w] = frequencies[w]/m
        if frequencies[w] >= self.threshold_max or frequencies[w] <= self.threshold_min:
            del frequencies[w]
    return frequencies
```

Once we have the frequency of all the words we will go to the text, sentence by sentence, and compute the sum of all the frequencies in that sentence. We will create a heap to keep the best and highest values on our list.

```
def summarize(self, sentence, lexical_chain, n):
    assert n <= len(sentence)
    word_sentence = [word_tokenize(s.lower()) for s in sentence]
    self.frequencies = self.return_frequencies(word_sentence, lexical_chain)
    ranking = defaultdict(int)
    for i,sent in enumerate(word_sentence):
        for word in sent:
            if word in self.frequencies:
                ranking[i] += self.frequencies[word]
                idx = self.rank(ranking, n)
    final_index = sorted(idx)
    return [sentence[j] for j in final_index]
```

We create a heap with all the sentences and its total frequency to keep the most important

```
def rank(self, ranking, n):
    return nlargest(n, ranking, key=ranking.get)
```

As a final result we will put in order the n most important sentences of our text that contain the largest amount of lexical chain and most frequency words in our text.