Parikshita Tripathi
A20327943

Lexical Chains:

Python Code Documentation

Reading the text file

Python commands file.read() was used to read the file.

The output of lexical chain is a list consisting of sublists by their lexical reltionships (i.e. Synonyms, hypernyms, hyponyms).
The output can be shown as: -

```
Chain 1 : ['beer(2)', 'pilsner(1)', 'ale(1)']
Chain 2 : ['miller(1)']
Chain 3 : ['snob(1)']
```

Tokenizing and Tagging

The sentences read from the file were tokenized using word_tokenize from NLTK.

For this assignment a part-of-speech tagger from NLTK was used to tag each word in the sentence we read. After that, only words with “NN”, “NNS”, “NNP” were extracted and used to genrate lexical chains.

Refer to function word_tokenize, pos_tag.

Code of the function used

```
# tokenize_data function takes the sentence, tokenize and tag it
# and return only words with tag as "NN"

def tokenize_data(sentence):
        token = word_tokenize(sentence)
        data_tags = pos_tag(token)
        data_nouns = [word for word,pos in data_tags if pos in ('NN', 'NNP')]
        for w, p in data_tags:
                if p == 'NNS':
                        if wn.synsets(w) == []:
                                wrd = w.rstrip('s')
                                if wn.synsets(wrd) != []:
                                        data_nouns.append(wrd)
                                        words = [w.lower() for w in data_nouns]
        return words
```

Parikshita Tripathi
A20327943

Chain Generation:

To generate chains, I used “Wu-Palmer Similarity” to calculate the score denoting the similarity in senses of the two words.

Syntax: dog.wup_similarity(cat)

Output: 0.857

```
def chain_gen(sentence):
   words = tokenize_data(sentence)
   lex_chain = []
   for i, word in enumerate(words):
       chain = []
       word_count = dict(Counter(words))
       for j in xrange(i+1, len(words)):
            # avoiding comparison of the word to itself
            if (i != j):
                 item1 = words[i] + "(" + str(word_count[words[i]]) + ")"
                 item2 = words[j] + "(" + str(word_count[words[j]]) + ")"

                 # genrating synsets of the consecutive words
                 if (words[i] != words[j]):
                      syn1 = wn.synsets(words[i])
                      syn2 = wn.synsets(words[j])

                      # compairing the synsets using wordnet.wup_similarity
                      for s1, s2 in product(syn1, syn2):
                           w = max((wn.wup_similarity(s1, s2) or 0) for s1, s2 in
                                                    product(syn1, syn2))

                           if item2 not in chain:
                               if (w >= 0.8) and (item2 not in chain):
                                    chain.append(item2)
                               if (w < 0.8) and (item1 not in chain):
                                    chain.append(item1)
                  lex_chain.append(chain)
return lex_chain
```

Final output genration:

To calculate the final chain, following function was written to remove redundancy and unwanted empty list in the output. The function below calls the chain_gen function (code above) and use the output to print the final result.

```
def lexical_chains(sentence):
    lex_chain = chain_gen(sentence)

    # loop to remove empty sublist
    for l in lex_chain:
         if l == []:
         lex_chain.remove(l)
```

Parikshita Tripathi
A20327943

```
# removes duplicate sublists
for i, c in enumerate(lex_chain):
     for j in xrange(i+1, len(lex_chain)):
         if j < len(lex_chain):
              if (lex_chain[i] == lex_chain[j]) :
                       lex_chain.pop(j)
              else:
                  lex_chain

# prints the final output
for i, l in enumerate(lex_chain):
     print "Chain", i+1, ":", l
```

Assignment 3: Lexical Chains

1. Create lexical chains based on the lexical relationships of synonymy, antonymy, and one level of hyper/hyponymy. In other words, if a noun is related by one of these relationships to an existing lexical chain, it should be added to it, otherwise it should start a new chain.

Output: -

Chain 1 : ['beer(2)', 'pilsner(1)', 'ale(1)']

Chain 2 : ['miller(1)']

Chain 3 : ['snob(1)']

2. Evaluate the results (qualitatively) and comment on how they might be improved.

- The relationship based on synonyms, and hyponyms are applied.
- The following relationaships are implemented: - synonyms, hypernyms and hyponyms
- And chains are grouped properly, the above output is a test output of a small file.

3. Implement one of the following enhancements:
   - Use more lexical relationships, including sibling hypo/hypernymy and meronym relationships.

   Done

4. Evaluate the results of your enhanced system, and compare them to the previous system.

The results were implemented on bigger files and below is the output: -

Parikshita Tripathi
A20327943

Chain 1 : ['world(1)', 'population(1)']

Chain 2 : ['pet(1)', 'cat(2)', 'mouse(1)', 'mice(1)']

Chain 3 : ['fact(1)', 'reason(1)']

Chain 4 : ['companionship(1)']

Chain 5 : ['reason(1)']

Chain 6 : ['enjoyment(1)']

Chain 7 : ['population(1)']

Chain 8 : ['house(2)', 'barn(1)']

Chain 9: ['contact(1)']

References:

http://effbot.org/zone/python-list.htm#creating

http://www.nltk.org/howto/wordnet.html

```
# Assignment 3: Natural Language Processing
# Author: Parikshita Tripathi
# Date: 10/26/2015

from nltk import pos_tag, word_tokenize
from nltk.corpus import wordnet as wn
from itertools import product
from collections import Counter

# tokenize_data function takes the sentence, tokenize and tag it
# and return only words with tag as "NN"
def tokenize_data(sentence):
    token = word_tokenize(sentence)
    data_tags = pos_tag(token)
    data_nouns = [word for word,pos in data_tags if pos in ('NN',
'NNP')]

    for w, p in data_tags:
        if p == 'NNS':
            if wn.synsets(w) == []:
                wrd  = w.rstrip('s')
                if wn.synsets(wrd) != []:
                    data_nouns.append(wrd)

    words = [w.lower() for w in data_nouns]
    return words

# chain_gen function calls tokenize_data and creates chain
# by compairing wup_similarity
def chain_gen(sentence):
    words = tokenize_data(sentence)
    lex_chain = []

    for i, word in enumerate(words):
        chain = []
        word_count = dict(Counter(words))

        for j in xrange(i+1, len(words)):
            # avoiding comparison of the word to itself
            if (i != j):
                item1 = words[i] + "(" + str(word_count[words[i]]) +
")"
                item2 = words[j] + "(" + str(word_count[words[j]]) +
")"

                # genrating synsets of the consecutive words
                if (words[i] != words[j]):
                    syn1 = wn.synsets(words[i])
                    syn2 = wn.synsets(words[j])
```

```
                    # compairing the synsets using
wordnet.wup_similarity
                    for s1, s2 in product(syn1, syn2):
                        w = max((wn.wup_similarity(s1, s2) or 0) for
s1, s2 in product(syn1, syn2))

                if item2 not in chain:
                        if (w >= 0.8) and (item2 not in chain):
                                chain.append(item2)
                        if (w < 0.8) and (item1 not in chain):
                            chain.append(item1)
        lex_chain.append(chain)

    return lex_chain

# improves redundancy of the chains
def lexical_chains(sentence):
    lex_chain = chain_gen(sentence)

    # loop to remove empty sublist
    for l in lex_chain:
        if l == []:
            lex_chain.remove(l)

    # removes duplicate sublists
    for i, c in enumerate(lex_chain):
        for j in xrange(i+1, len(lex_chain)):
            if j < len(lex_chain):
                if (lex_chain[i] == lex_chain[j]) :
                    lex_chain.pop(j)
                if (list(set(lex_chain[i]).union(lex_chain[j]) ==
lex_chain[i])) or (list(set(lex_chain[i]).union(lex_chain[j]) ==
lex_chain[j])):
                    lex_chain =
list(set(lex_chain[i]).union(lex_chain[j]))
                    lex_chain.pop(j)
                else:
                    lex_chain

    # prints the final output
    for i, l in enumerate(lex_chain):
        print "Chain", i+1, ":", l


#sentence = "I like beer. Miller just launched a new pilsner. But,
because I'm a beer snob, I'm only going to drink pretentious Belgian
ale."
sentence = open('testFile1.txt')
file = sentence.read()
lexical_chains(file)
```

```
#lexical_chains(sentence)
```