NLP Lab Session Week 3
September 11, 2013

Starting a Python and an NLTK Session

Open a Python 2.7 IDLE (Python GUI) window by going to All Programs->Python 2.7 -> IDLE (Python GUI).

In this lab session, we will work together through a series of small examples using the IDLE window and that will be described in this lab document. However, for purposes of using cut-and-paste to put examples into IDLE, the examples can also be found in a set of python files on the iLMS system, under Resources. I loaded this file both with the extension .py and .txt to see which one was best for you to download from Blackboard.

Download either: LabWeek3examples.py or LabWeek3examples.txt
and save it in a folder on your H: drive where you will keep materials for this class. If you downloaded the one with extension .txt, rename it to have .py.

Open an IDLE window. Use the File-> Open to open the labweek3examples.py file. This should start another IDLE window with the program in it. Each example line can be cut-and-paste to the IDLE window to try it out.

Getting Started in Python and NLTK

First let's load nltk.

```
>>> import nltk
```

Next, we want to set up the emma text again for processing. Go to the file with LabWeek3examples.py and copy and paste the following lines to get started. This gets the text of the book Emma, separates it into tokens with the wordpunct tokenizer, and converts all the characters to lower case.

Notes: Paste one line at a time! Any line that starts with the character # is a python comment and doesn't need to be pasted. (This can be any text that explains to a person what is going on. Python doesn't try to process it.)

```
>>> from nltk import FreqDist
>>> print nltk.corpus.gutenberg.fileids( )
>>> file0 = nltk.corpus.gutenberg.fileids( ) [0]
>>> emmatext = nltk.corpus.gutenberg.raw(file0)
>>> emmatokens = nltk.wordpunct_tokenize(emmatext)
>>> emmawords = [w.lower( ) for w in emmatokens]
```

For purposes of the lab, we create a list with only the first 101 words that follow the title and author.

```
>>> shortwords = emmawords[11:111]
>>> shortwords
```

As before, we create a frequency distribution of the words, using the NLTK FreqDist module/class.

```
>>> shortdist = FreqDist(shortwords)
>>> shortdist.keys( )

>>> for word in shortdist.keys():
      print word, shortdist[word]
```

Again note the special syntax of Python for a multi-line statement: The first line must be followed by an extra ":", and each succeeding line must be indented by some number of spaces (as long as they are all indented by the **same** number of spaces.)

More Specialized Frequency Distributions (What is a word?)

We note that the WordPunct tokenization produces tokens that have special characters in them. Let's remove all the tokens that have special characters and leave only tokens that consist of all alphabetic characters. (Note that this is not realistic. In practice, we would like to leave some special characters, such as words with embedded hypens, for example "lower-case" and "need-based". We'll be writing a more detailed tokenizer in a later lab.)

We'll use a regular expression that matches any token that contains a non-alphabetical character. We'll be covering regular expressions in class next week, but for now, we can just use this one.

```
>>> import re
# this regular expression pattern matches any characters followed by a non-alphabetical
#   lower-case character [^a-z] followed by some more characters
>>> pattern = re.compile('.*[^a-z].*')
```

Apply the pattern to the string '-' to see if it matches. Note that the result of the .match function can be used as a Boolean expression, either true or false, so you can use it in an "if" test.

```
>>> nonAlphaMatch = pattern.match('-')

#  if it matched, print a message
>>> if nonAlphaMatch: 'matched non-alphabetical'
```

In Python, we can make a function that can take any argument, process it and return a result. For an example function, we make a function called "alphaFreqDist" that takes a list of words as an argument and returns a Frequency Distribution which only contains words with all alphabetical characters.

Copy and paste this entire function definition into the Idle window.

```
>>> def alphaFreqDist(words):
   # make a new frequency distribution called adist
   adist = FreqDist()
   # define the regular expression pattern to match non-alphabetical tokens
   pattern = re.compile('.*[^a-z].*')
   # for every token, if it doesn't match the non-alphabetical pattern
   #   add it to the frequency distribution
   for word in words:
      if not pattern.match(word):
        adist.inc(word)
   # return the frequency distribution as the result
   return adist
```

Apply the new function to shortwords and look at the first 50 words:

```
>>> adist = alphaFreqDist(shortwords)
>>> adist.keys()[:50]
```

Print out the 30 top words:

```
>>> for key in adist.keys()[:30]:
  print key, adist[key]
```

For comparison, we can make a frequency distribution of all the emma words and display the top alphabetical words.

```
>>> bigadist = alphaFreqDist(emmawords)
>>> bigadist.keys()[:99]
>>> for key in bigadist.keys()[:30]:
   print key, bigadist[key]
```

Our next step is to remove some of the common words that appear with great frequency. This is usually done by making a list of the words to remove, known as a *stop word* list. Every token is compared with the stop word list and not entered into the frequency distribution is it appears on the list.

For purposes of the lab, we'll just define our stopwords to be a short list that we observed were frequent in Emma. In the future, we'll use a list of stopwords from a file. Note that what the stopword list should be will depend on the analysis that is using the word frequency list. An example is the inclusion of pronouns; if you are looking at the frequencies of topic words, you remove pronouns, but if you are looking at words contributing to literary style, you would include them.

```
>>> stopwords = ['to', 'be', 'of', 'the', 'in', 'it', 'was', 'i', 'am', 'she', 'had', 'been', 'is', 'have',
'could', 'not', 'her', 'he', 'do', 'and', 'would', 'such', 'a', 'his', 'must']
```

Test if a word is in a list by using the Python keyword "in":

```
>>> word = 'the'
>>> if word in stopwords:
   print 'Stop!'
```

Now we define a function to make a frequency distribution from a list of tokens that has no tokens that contain non-alphabetical characters or words in the stopword list. We will pass the stop word list into the function as a second argument. (Now the function is defined to take two arguments and return one result, which is a frequency distribution.)

```
>>> def alphaStopFreqDist(words, stoplist):
   # make a new frequency distribution called asdist
   asdist = FreqDist()
   # define the regular expression pattern to match non-alphabetical tokens
   pattern = re.compile('.*[^a-z].*')
   # for every token, if it doesn't match the non-alphabetical pattern
   #   and if it is not on the stop word list
   #     add it to the frequency distribution
   for word in words:
      if not pattern.match(word):
         if not word in stoplist:
            asdist.inc(word)
   # return the frequency distribution as the result
   return asdist
```

Apply the new function to shortwords and the short stopword list that we defined above.

```
>>> asdist = alphaStopFreqDist(shortwords, stopwords)
>>> asdist.keys()[:50]
```

Print out the 30 top words:

```
>>> for key in asdist.keys()[:30]:
  print key, asdist[key]
```

Apply the function to emmawords

```
>>> bigasdist = alphaStopFreqDist(emmawords, stopwords)

>>> bigasdist.keys()[:99]

>>> for key in bigasdist.keys()[:30]:
  print key, bigasdist[key]
```

We can see that our short stop word list is not adequate to remove a lot of the non-content bearing words. We'll use bigger stop word lists in the future.

Bigram Frequency Distributions

Another way to look for interesting characterizations of a corpus is to look at pairs of words that are frequently collocated, that is, they occur in a sequence called a bigram.

We can define a function that takes a list of words (and a stoplist) and makes a frequency distribution that consists of pairs of words. For convenience in printing out the results, instead of actually making a "pair" or "tuple", which Python would represent as (firstword, secondword), we'll make a string that has the two words separated by a space "firstword secondword".

In this version of the bigram frequency distribution function, we will also make sure that we restrict our words to those that occur in a unigram/word frequency distribution without non-alphabetical characters and stop words. Note that any word frequency distribution function could be substituted.

```
>>> def bigramDist(words, stoplist):
   biDist = FreqDist()
   # use one of the word frequency distributions to limit the tokens
   uniDist = alphaStopFreqDist(words, stoplist)
   # loop through the words in order, looking at all pairs of words
   for i in range(1, len(words)):
      if words[i-1] in uniDist and words[i] in uniDist:
         biword = words[i-1] + ' ' + words[i]
         biDist.inc(biword)
   return biDist
```

Try out our bigram function on shortwords and emmawords.

```
>>> shortbidist = bigramDist(shortwords, stopwords)
>>> shortbidist.keys()

>>>emmabidist = bigramDist(emmawords, stopwords)

>>> for key in emmabidist.keys()[:30]:
  print key, emmabidist[key]
```

These pairs of words seem to show more information about the contents of the corpus. But I think that it will be even better when we use a more extensive stop word list.

Mutual Information

Look at the paper by Church and Hanks and observer their definition of the Association Ratio. Note that technically the original information theoretic definition of mutual information allows the two words to be in either order, but that the association ratio defined by Church and Hanks requires the words to be in order from left to right wherever they appear in the window. I have

written a function for the association ratio with a window of 2, which means that the word pairs are all bigrams. I called it mutualinfo after the modern practice of calling this mutual information. There is another argument to the function which is the threshold, where no bigrams are included that contain words whose frequency if less than or equal to the threshold.

The definition of the mutual information score requires us to take a logarithm of the ratio, so we import that from the math module.

```
>>> from math import *

# Mutual Information (Association Ratio) function
# Parameters:
#    words should be a list of tokens (can be lower-cased or not)
#    stoplist should be a list of stop words, but can be empty []
#    threshold should be the minimum frequency of individual words
#        e.g. threshold of 2 omits words with frequency 1 (only occurs once)
# Returns a frequency distribution with bigram keys and mutual information scores as values
>>> def mutualinfo(words, stoplist, threshold):
    # make an empty frequency distribution
    assocDist = nltk.FreqDist()
    # frequencies of words using the unigram freq dist with alphabetical and no stopwords
    uniDist = alphaStopFreqDist(words, stoplist)
    # compute frequencies of word pairs in window of length 2
    for i in range(1, len(words)):
        if (uniDist[words[i-1]] > threshold) and (uniDist[words[i]] > threshold):
            biword = words[i-1] + ' ' + words[i]
            assocDist.inc(biword)
    # compute mutual information ratio for each word pair
    # make a new frequency distribution whose values will be mutual information scores
    arDist = nltk.FreqDist()
    N = len(words)
    # loop over all the bigram strings that are keys in the bigram distribution
    for wordstring in assocDist.keys():
        wordlist = str.split(wordstring, ' ')
        w0 = wordlist[0]
        w1 = wordlist[1]
        ar = (assocDist[wordstring] * N * N) / (1.0 *uniDist[w0] * uniDist[w1])
        ar_log2 = log(ar, 2)
        arDist[wordstring] = ar_log2
    return arDist
```

Let's call the function with an empty stop word list and a threshold of 2:

```
>>> MIdist = Bigram.mutualinfo ( emmawords, [] , 2)
```

Try looking at the keys and mutual information scores for the top 20 scoring bigrams.

```
>>> for pair in MIdist.keys() [:30]:
     print pair, MIdist[pair]
```

Note the differences between the mutual information lists and the bigram lists that we got previously.

The above examples used a threshold of 2, but Church and Hanks actually recommend a threshold of 5.

In next week's lab, we'll show how to store these functions in a module to make them easier to define with copy/pasting the definitions.

Exercise for Week 3:

For this exercise, you may work in groups of 2-3, or you can choose to do your own work.

- Choose a file that you want to work on, either one of the files from the book corpus, or one from the Gutenberg corpus.
-
- Run the different frequency distribution functions on your corpus and look at the top 20 keys from each of them: FreqDist, alphaFreqDist, alphaStopFreqDist, and bigramDist. Do you see any improvements that should be made to these distributions?

To complete the exercise, choose one of your top 20 frequency lists to report to show to the class. Write an introductory sentence of paragraph telling what text you chose and what bigram distribution function you chose. Put this and the frequency list in a discussion posting in the blackboard system under the Discussions tab.