NLP Lab Session Week 3
Bigram Frequencies and Mutual Information Scores in NLTK
September 16, 2015

Starting a Python and an NLTK Session

Open a Python 2.7 IDLE (Python GUI) window or a Python interpreter window.

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 Blackboard, in the Lab Materials folder.

Download LabWeek3examples.txt

and save it in a folder where you keep materials for this class. Each example line can be cut-and-paste to the IDLE or Python interpreter 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.txt and copy and paste the following lines to get started. This gets the text of the book Emma, separates it into tokens with the word 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.word_tokenize(emmatext)
>>> emmawords = [w.lower( ) for w in emmatokens]
```

For purposes of the lab, we create a list with only the first 100 words that follow the title and author. We first observe that the title and author take up the first 11 tokens.

```
>>> emmawords[:110]
>>> shortwords = emmawords[11:110]
>>> 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.)

Note on NLTK 3.x: If at home, you installed a later version of NLTK, the FreqDist will not be in order by frequency. Instead you can get a list of items, which are the words and frequencies in order:

```
>>> ndist = FreqDist(shortwords)
>>> nitems = ndist.most_common(30)
>>> for item in nitems:
   print item[0], item[1]
```

More Specialized Frequency Distributions (What is a word?)

We note that the word tokenization produces tokens that have special characters in them. Let's remove all the tokens that have only special characters. Note that this leaves some special characters, such as words with embedded hypens, for example "lower-case" and "need-based".

We'll use a regular expression that matches any token that contains all 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 word that contains all non-alphabetical
#   lower-case characters [^a-z]+
# the beginning ^ and ending $ require the match to begin and end on a word boundary
>>> 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 “alpha_filter” that takes word as an argument and returns True if it contains all non-alphabetical characters and False otherwise.

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

```
>>> # function that takes a word and returns true if it consists only
#   of non-alphabetic characters

def alpha_filter(w):
  # pattern to match a word of non-alphabetical characters
    pattern = re.compile('^[^a-z]+$')
    if (pattern.match(w)):
        return True
    else:
        return False
```

Apply the new function to shortwords to include only those that don’t match the filter:

```
>>> alphashortwords = [w for w in shortwords if not alpha_filter(w)]
>>> alphashortwords
```

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 use the stop word list given by NLTK. In the future, we’ll use a list of stopwords from a file if you want to design your own custom stop word list. 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 = nltk.corpus.stopwords.words('english')
>>> len(stopwords)
>>> stopwords
```

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.)

We can also filter our shortwords list to not include any stopwords that we defined above.

```
>>> stoppedshortwords = [w for w in shortwords if not w in stopwords]
>>> stoppedshortwords
```

Note that this big stop word list removes a lot of the non content-bearing words in the list.

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.

First, we just simply look at the bigrams that can be defined.

```
>>> shortbigrams = list(nltk.bigrams(shortwords))
>>> shortbigrams[:20]
```

(The nltk.bigrams() function returns a generator, but we can turn it into a list by applying the type 'list' to it as a function.)

But we will obtain a lot more functionality by using the functions from the nltk.collocations package. One of the places to obtain information about this package is in this section of the NLTK documentation called the HOWTOS. We note that there are also Trigram functions in addition to the Bigram functions shown here.

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

To start using bigrams, we import the collocation finder module.

```
>>> from nltk.collocations import *
```

Next, for convenience, we define a variable for the bigram measures.

```
>>> bigram_measures = nltk.collocations.BigramAssocMeasures()
```

We start by making an object called a BigramCollocationFinder. The finder then allows us to call other functions to filter the bigrams that it collected and to give scores to the bigrams. We start by scoring the bigrams by frequency:

```
>>> finder = BigramCollocationFinder.from_words(shortwords)
>>> scored = finder.score_ngrams(bigram_measures.raw_freq)
>>> scored
```

The result from the score_ngrams function is a list consisting of pairs, where each pair is a bigram and its score. The raw_freq measure returns frequency as the ratio of the count of the bigram over the count of the total bigrams.

```
>>> type(scored)
>>> first = scored[0]
>>> type(first)
>>> first
```

We can see that these scores are sorted into order by decreasing frequency.

```
>>> for bscore in scored[:30]:
   print bscore
```

For any finder, we can also apply various filter functions. First let's apply our alpha_filter that we created earlier. It uses a filter that is applied to the individual words. Note that the function apply_word_filter changes the bigram collocation in the variable "finder", and any function which takes a word parameter and returns True or False as a result can be used.

```
>>> finder.apply_word_filter(alpha_filter)
>>> scored = finder.score_ngrams(bigram_measures.raw_freq)
>>> for bscore in scored[:30]:
   print bscore
```

Next we can filter out the stopwords if we wish. Note that the lambda operates like a function definition "on-the-fly", i.e. without a function name.

```
>>> finder.apply_word_filter(lambda w: w in stopwords)
>>> scored = finder.score_ngrams(bigram_measures.raw_freq)
>>> for bscore in scored[:20]:
   print bscore
```

Another filter would remove words that only occurred with a frequency over some minimum threshold.

But to show its effect, we first start over with a newly defined finder.

```
>>> finder2 = BigramCollocationFinder.from_words(shortwords)
>>> finder2.apply_freq_filter(2)
>>> scored = finder2.score_ngrams(bigram_measures.raw_freq)
>>> for bscore in scored[:20]:
   print bscore
```

Another word level filter would be to remove words whose length is small. Or we can apply filters that work on both words of the bigram. Suppose that we want to filter out bigrams in which the first word's length is less than 2. Then we could use an ngram filter that is applied to both words of the bigram.

```
>>> finder2.apply_ngram_filter(lambda w1, w2: len(w1) < 2)
>>> scored = finder2.score_ngrams(bigram_measures.raw_freq)
>>> for bscore in scored[:20]:
   print bscore
```

Mutual Information and other scorers

Recall that Mutual Information is a score introduced in the paper by Church and Hanks, where they defined it as an 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

In NLTK, the mutual information score is given by a function for Pointwise Mutual Information, where this is the version without the window.

```
>>> finder3 = BigramCollocationFinder.from_words(shortwords)
>>> scored = finder3.score_ngrams(bigram_measures.pmi)
>>> for bscore in scored[:s0]:
   print bscore
```

When you apply the Mutual Information score to small documents, the results don't really make sense. In particular, here all our scores are the same. It is recommended to run the PMI scorer with a minimum frequency of 5, which will make more sense on very large documents.

See the documentation for other scoring functions.

Bigrams from the entire Emma book

Finally, let's put together our work on bigrams to display bigrams from the entire emma book. We'll repeat many of our steps for emmawords, instead of shortwords.

These two scorers show the bigrams in order by frequency.

```
>>> finder = BigramCollocationFinder.from_words(emmawords)
>>> scored = finder.score_ngrams(bigram_measures.raw_freq)
>>> len(scored)
67625
>>> for bscore in scored[:30]:
   print bscore
```

Apply the non-alphabetical and stopword filters, noting that the filters substantially reduce the numbers of bigrams scored.

```
>>> finder.apply_word_filter(alpha_filter)
>>> finder.apply_word_filter(lambda w: w in stopwords)
>>> scored = finder.score_ngrams(bigram_measures.raw_freq)
>>> len(scored)
16070
Top 30 bigrams by frequency:
>>> for bscore in scored[:30]:
   print bscore
```

This scorer shows the bigrams in order by Pointwise Mutual Information.

```
>>> scored = finder.score_ngrams(bigram_measures.pmi)
>>> len(scored)
16070
```

Top 30 bigrams by pointwise mutual information scores:

```
>>> for bscore in scored[:30]:
   print bscore]
```

Without a minimum frequency filter, the PMI scores tend to score highly on bigrams that contain unique words, which includes words that have errors in tokenization. So instead, we further filter the bigrams by frequency and then apply the PMI scores.

```
>>> finder.apply_freq_filter(5)
>>> scored = finder.score_ngrams(bigram_measures.pmi)
>>> for bscore in scored[:30]:
   print bscore
```

Looking ahead to the first HomeWork assignment, running the bigram frequencies and PMI scores is an example of what you'll be exploring for that assignment.

Exercise for Week 3:

For this exercise,

- Choose a file that you want to work on, either one of the files from the book corpus, or one from the Gutenberg corpus.
-
- Make a bigram finder and experiment with whether to apply the filters or not . Run the scoring with both the raw frequency and the pmi scorers and compare results.

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 filters and scorer you used. Put this and the frequency list in a discussion posting in the blackboard system under the Discussions tab.