**NLP Lab Session Week 2**
**Bigram Frequencies and Mutual Information Scores in NLTK**
September 07, 2017

## Starting a Jupyter Notebook Python Session

In this week, we will start using Jupyter Notebook for our Python interpreter. This will have the advantage of being able to more easily go back and execute code again.

For this lab session, we will work together through a series of small examples again. The examples are published in two formats: a notebook session and a python file. The notebook session is a json representation of the code, where you can import it directly into a jupyter session. The python file gives an easily readable version of the code. Either can be used to copy/paste examples into the interpreter.

Download LabExamplesWeek2Bigrams.ipynb and LabExamplesWeek2Bigrams.py.

and save it in a folder where you keep materials for this class. Each section of examples can be copy/pasted to a jupyter session.

## Getting Started in Python and NLTK

First we go to a command prompt window (in Windows) or a terminal session (Macs) and go to the folder where we keep our class lab examples. For example:

$ cd nlpclassfall2017/labs

Then type at the prompt:

labs$ jupyter notebook

This brings up a browser window in your default browser and shows the contents of your labs folder. This will be the folder where jupyter will save your python sessions. Now either load the LabExamplesWeek2Bigrams.ipynb session to execute the cells or open the LabExamplesWeek2Bigrams.py program to copy/paste the python code.

In the latter case, open a new session for Python 3. Note that Jupyter notebook has spaces called cells where you can type one or more python statements. Then you can execute the cell (where a keyboard shortcut is “shift enter”).

```
import nltk
from nltk import FreqDist
```

Now we want to set up the emma text again for processing by repeating steps that we did last week. This gets the text of the book Emma, separates it into tokens with the word tokenizer, and converts all the characters to lower case.

```
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]
# show the number of words and print the first 110 words
print(len(emmawords))
print(emmawords[ :110])
```

As before, we create a frequency distribution of the words, using the NLTK FreqDist module/class, and we show the 30 top frequency words.

```
ndist = FreqDist(emmawords)
nitems = ndist.most_common(30)
for item in nitems:
   print (item[0], '\t',item[1])
```

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.)

**Digression on different tokenizations**

So far, we loaded the text from the Gutenberg files as raw text and called the NLTK word_tokenize function. But, in fact, there is another version of the text that is already tokenized by some other function.

We can get this different version of the words by using a function called *words(fields = [f1, f2, f3])* that returns the words of the specified fileids (please refer to the tables at the end of this lab). So we can do something like this.

```
emmawords2 = nltk.corpus.gutenberg.words('austen-emma.txt')
emmawords2lowercase = [w.lower() for w in emmawords1]
```

Do you think the word lists that emmawords and emmawords2lowercase contain should be identical? Try this out and see if this is what you expected:

```
len(emmawords)
len(emmawords2lowercase)
```

Now try printing some of the words. What difference do you see?

```
emmawords[:50]
emmawords2lowercase[:50]
```

**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.

```
# 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 emmawords to include only those that don't match the filter:

```
alphaemmawords = [w for w in emmawords if not alpha_filter(w)]
print(len(alphaemmawords))
print(alphaemmawords[:100])
```

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.

Also, you should check that the tokenization in the stopword list matches the tokenization in your text.

```
stopwords = nltk.corpus.stopwords.words('english')
print('number stopwords:', len(stopwords))
print(stopwords)
```

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

```
word = 'the'
if word in stopwords:
   print (word + ' is a stopword!')
```

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.

```
stoppedemmawords = [w for w in alphaemmawords if not w in stopwords]
print(len(stoppedemmawords))
```

Now we can remake our frequency distribution with our new filtered word list.

```
emmadist = FreqDist(stoppedemmawords)
emmaitems = emmadist.most_common(30)
for item in emmaitems:
  print(item)
```

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.
```
emmabigrams = list(nltk.bigrams(emmawords))
print(emmabigrams[: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 by calling the score_ngrams function with the raw_freq scoring measure.

** Note that you must us the entire list of emmawords before any filtering or the raw bigrams will not be correct. Start with all the words and then run the filters in the bigram finder.

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

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.

```
print(type(scored))
first = scored[0]
print(type(first), first)
```

We can see that these scores are sorted into order by decreasing frequency. The scores are the frequencies of the bigrams, normalized to fractions by the total number of bigrams.

```
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[:30]:
   print (bscore)
```

There are other possible filter effects, for example, 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(emmawords)
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(emmawords)
scored = finder3.score_ngrams(bigram_measures.pmi)
for bscore in scored[:30]:
   print (bscore)
```

But when you apply the Mutual Information score to small documents or documents with rare words, 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.

```
finder3.apply_freq_filter(5)
scored = finder3.score_ngrams(bigram_measures.pmi)
for bscore in scored[:30]:
   print (bscore)
```

See the documentation for other scoring functions.

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 2:**

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.

**Appendix:** Functions for Frequency Distributions and Corpus Readers

**Table 3.1**: Functions Defined for NLTK's FreqDist (from NLTK book chapter 1)

| Example | Description |
|---|---|
| `fdist = FreqDist(samples)` | create a frequency distribution containing the given samples |
| `fdist[sample] += 1` | increment the count for this sample |
| `fdist['monstrous']` | count of the number of times a given sample occurred |
| `fdist.freq('monstrous')` | frequency of a given sample |
| `fdist.N()` | total number of samples |
| `fdist.most_common(n)` | the `n` most common samples and their frequencies |
| `for sample in fdist:` | iterate over the samples |
| `fdist.max()` | sample with the greatest count |
| `fdist.tabulate()` | tabulate the frequency distribution |
| `fdist.plot()` | graphical plot of the frequency distribution |
| `fdist.plot(cumulative=True)` | cumulative plot of the frequency distribution |
| `fdist1 \|= fdist2` | update `fdist1` with counts from `fdist2` |
| `fdist1 < fdist2` | test if samples in `fdist1` occur less frequently than in `fdist2` |

**Table 1.3**: Basic Corpus Functionality defined in NLTK: more documentation can be found using `help(nltk.corpus.reader)` and by reading the online Corpus HOWTO at `http://nltk.org/howto`. – table from NLTK book Chapter 2

| Example | Description |
|---|---|
| `fileids()` | the files of the corpus |
| `fileids([categories])` | the files of the corpus corresponding to these categories |
| `categories()` | the categories of the corpus |
| `categories([fileids])` | the categories of the corpus corresponding to these files |
| `raw()` | the raw content of the corpus |
| `raw(fileids=[f1,f2,f3])` | the raw content of the specified files |
| `raw(categories=[c1,c2])` | the raw content of the specified categories |
| `words()` | the words of the whole corpus |
| `words(fileids=[f1,f2,f3])` | the words of the specified fileids |
| `words(categories=[c1,c2])` | the words of the specified categories |
| `sents()` | the sentences of the whole corpus |
| `sents(fileids=[f1,f2,f3])` | the sentences of the specified fileids |
| `sents(categories=[c1,c2])` | the sentences of the specified categories |
| `abspath(fileid)` | the location of the given file on disk |

| Example | Description |
| --- | --- |
| `encoding(fileid)` | the encoding of the file (if known) |
| `open(fileid)` | open a stream for reading the given corpus file |
| `root` | if the path to the root of locally installed corpus |
| `readme()` | the contents of the README file of the corpus |