Counting words and n-grams

We will set up the emma text again for processing. The following lines get the text of the book Emma, separates it into tokens with the wordpunct tokenizer, and converts all the characters to lower case, as done in the last lab.

Notes: Any line that starts with the character # is a python comment.

```
>>> 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, as follows.

```
>>> 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 hyphens, for example "lower-case" and "need-based").

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 lowe-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, we already wrote a function called `alphaFreqDist` that took a list of words as an *argument* and *returned* a Frequency Distribution which only contains words with all alphabetical characters.

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 stopword list into the function as a second argument. (Now the function is defined to take two arguments and return one result.)

```
>>> 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 stopword 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()
uniDist = alphaStopFreqDist(words, stoplist)
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. It will be even better when we use a more extensive stop word list.

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. 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?