Jordan Barber
Assignment 2
April 27, 2015

I chose Python as my language of choice because the library support is so extensive, and from prior classes I know that it is a common choice for natural language processing problems. I also work best when I can use the interpreter to easily inject test variables and run snippets at a time. I approached my script by looking at a single text file for unigrams only, in order to iteratively (and slowly) add additional layers of complexity to handle the bigrams and trigrams. I also did this because I was not totally sure that there was a single library for all of them.

After getting the first text file into a string variable, it occurred to me that stopping and stemming may require the words to be broken into a list of individual strings. Fortunately, the requirement to *tokenize* text seems necessary for most natural language processing, so finding a solution was pretty easy. While looking into tokenizing solutions, it became apparent that the NLTK platform was the most comprehensive set of tools and would cover most of the processing needs for the script.

From the NLTK package, I utilized the Porter Stemmer function, which appears to be the most popular stemming method and handles the most common word endings. I also used the RegexpTokenizer, bigrams and trigrams functions. I used the Python library's stopwords collection, because it had fewer stop words and I wanted to be a little conservative on words that I remove before using something more aggressive.

Producing a working script for the unigrams was much easier than the bigrams and trigrams I added later. My initial script for the unigrams converted them into a dictionary so that the words were the key and their frequency was the value. This dictionary was then looped through and written to a csv. This created some problems when coding the bigrams, because the Counter method I used didn't interact well with lists of multiple words per string. So at this point I decided I needed a slightly different approach, but rather than refactoring the unigram case so that all ngrams would use the same code, I branched the ngrams into three different variables after the tokens were stopped and stemmed. This is slightly lazy and a little slower, but because the unigram code was essentially "done," it made more sense to tackle the rest separately.

NLTK has a bigram and trigram method that creates a bigram and trigram generator object. However, those objects need to be iterated through in order to be read, so I then looped through them to create a bigram list and trigram list. This felt a little roundabout, but the NLTK documentation on generator objects wasn't entirely clear on how to output their data. In any case, NLTK also had a very useful fdist function that accepted lists and output a dictionary-like generator object that paired each bigram or trigram with their frequency in the document. I then wrote all three separate dictionaries to a csv file.

To compute the sum ngrams for all documents, I inserted a variable that appended each document's ngram lists. After the documents were completed, those sum ngram lists were run through the process as the documents, outputting to a total csv file.

These kinds of data collection and processing scripts can be challenging in that reading API or library documentation usually takes more time than writing the actual code. It can be frustrating to connect so many disparate libraries together, converting variable types into a form that the next function accepts, etc. Python is invaluable for this, not only because there are so many libraries available (and are usually well documented) and the interpreter you to quickly test.