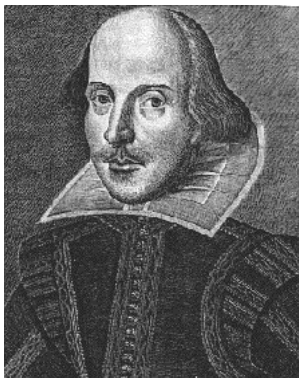
CS2204
Project No. 6: Shake-n-Bacon

Purpose:
Gain experience in using Python sets and dictionaries.



Background:
William Shakespeare is arguably one of the most notable authors of the English language. The ultimate wordsmith, many of the phrases he coined remain in common use today and his plays are regularly performed around the world. But some have doubts that the bard actually wrote the texts commonly attributed to him.

The most common theory is that while it is true Shakespeare was an actor and his name was connected with the works, he did not have the education required to write the plays and sonnets. Depending on which of the many theories you read, many different authors are put forth as ghost writers for his works. One of the first and most popular candidates is Sir Francis Bacon. In completing this assignment you will produce information that will support or detract from the simplified theory that Bacon wrote Shakespeare's works.

First, as on any good expedition, you need to bring the right tools. In evaluating authorship we will be using 2 major tools:

- Word usage comparison
- Word Stemming

Word Usage Comparison

In general, each person has their own unique writing "signature" that is evident in the words and grammatical structure they use. Due to differences in vocabulary, speech patterns, and other environmental factors, these signatures can be identified by a light form of statistical analysis. Basically, you count each time a word is used in a document and compare the percentile appearance to the same word in another document. If the word is used a similar amount in both documents then it is more likely that the author was the same person. By repeating this process for all unique words in the document, a conclusion about authorship of a new document can be made with higher certainty.

A few limits are imposed on the comparison process to make it more accurate. First, many words are commonly used within the language itself or the files under consideration. For example, in English you would expect the word "the" to have a high count in most documents. You will be provided with a list of words that are to be completely ignored in processing the documents. Second, some words are very rare in one of the files and provide very little information about authorship differences. These words tend to be proper names or obscure words that are not likely to be shared even by works from the same author. We will not compare the counts of these words, but their appearances still count towards the overall total number of words in the file.

Word stemming

The second tool that you will use is called word stemming. Word stemming is a very difficult problem to solve programmatically and continues to intrigue those in the study of Natural Language Processing (NLP). The primary purpose of word stemming is to recognize when two words share a common meaning or root. A good example of stemming is the tense of a verb. "Run", "running", and "ran" all share the same root ("run"), but our program as described so far will count them as different words. The history of English as an amalgamated language makes generalizing rules for stemming difficult. If we tried to write

a program that stemmed all words that ended with "nning" to a single "n" we would get the following transformations:

```
Running -> run   (OK)
Spanning -> span (OK)
Inning -> in     (Bad, "inning" cannot be stemmed here)
```

There are similar exceptions for many of the other simple rules we might think up.

You are welcome to explore the concept of stemming further outside of the class, and there are a number of more interesting algorithms commonly used including porter stemming (http://tartarus.org/~martin/PorterStemmer/). For this assignment though, we will use a very primitive form of stemming by shortening every word found to STEM_LENGTH which is declared as 6 characters. More complex forms of word stemming are used in spell checkers and search engines.

The Assignment:

In this assignment, we are going to create a program that will count the number of occurrences of words in a file, while ignoring words that are considered to be "common". Using such a list, we will compare two files to attempt to answer the authorship of Shakespeare's works. This program is actually quite easy to implement when using Python dictionaries and sets. We will use the **set** container to hold the set of common words we are not interested in counting. We will use the **dictionary** container to hold pairs that represent a word-to-count mapping (i.e.; each word will map to an integer that represents how many times that word appeared in the file), as well as word-to-distance mapping (i.e., the distance between two documents for a given word).

Step I:

Create a new project for this assignment. Once the project is created, copy the supplied `word_count.py` file into the project directory and add it to the project. Also add the "texts" folder which contains the documents we wish to compare.

Take a moment to view the supplied code. The main method consists mostly of some function calls. To keep things simple, we have hard-coded the file names of the files we want to analyze [thus you will need to edit the code to analyze a different pair of files]. The input files will be located in the subdirectory "texts". The output file will go into the directory above "texts".

Step II:

Implement the `read_common_words()` function. The function needs to create a set of common words (as strings) to ignore in later processing. First, open the common words file. After opening the file, read all the words from the file, sanitize them (removing whitespaces, etc.), and insert each word into the set. When done processing the file, close the file and return the set of words.

Step III:

Implement the `process_file()` function. This function takes two arguments: the name of the file to be processed and the set of words that are considered "common". This function needs to create a dictionary object, which will map a string (word) to an integer (count). This function then opens the specified file and reads and processes all the words. This process has three steps for every word:

1. Convert all letters to lower case and remove punctuation & other non-alphabetic characters.
2. Use our simple form of stemming: truncate any word that is longer than 6 letters to the first six letters.
3. Determine if the resulting word is in the set of common words. If it is a common word, we ignore it. Similarly, we ignore any resulting word that is an empty string. If the resulting word is not a common word and is non-empty, we find its current count in the dictionary and increment it by one. If the word is not yet in the dictionary, add it to the dictionary and set the count to one.

When done processing all the words in the file, we close the input file and return the dictionary to the caller.

Step IV:
Implement the `compare_texts()` function. This function takes two parameters: the dictionaries of the two files to compare. The function returns a new dictionary of words common to the two parameter dictionaries. The value associated with each word in the new dictionary is the squared distance between the two files (details below). The function needs to compare the files as described below. Each word will have a percentile measurement (or *score*) for its appearance found by dividing the count for that word by the sum of all counts in the dictionary. The percentile measure helps to normalize the information so we can more accurately compare texts with largely different sizes.

To compare the two texts, we will be using a version of Euclidean distance. In this case, each dictionary is an n-element vector where each word has an entry. To be fair in computing the distance we want to ignore words that appear very rarely and will only compare words that appear in *both* files with a percentile score such that: `0.001 < score`. So, for each word that appears sufficiently in both files, add the word to the result dictionary and associate with it the value of the square of the difference (see the following expression):

$$(score1 - score2)^2$$

Where `score1` and `score2` are the percentile scores for the word in each dictionary. The final dictionary to return should have the frequently occurring common words as keys and their calculated distances as their values. We will use this dictionary to calculate the total distance between the files in a later step.

Step V:
Implement the `report_difference()` function. This function takes two parameters: the dictionary calculated in step IV and an output filename. The function should iterate over the dictionary and sum all the word distances to compute a total file distance. As you iterate, you should be printing out the contents of the dictionary to the output file in **sorted order**. Note that the dictionary is not stored in sorted order. After the sum has been calculated, your last write to the file should be the computed vector distance between the two files. You will also return this value from the function so that the calling function can print this line to the console as well. Use the following format:

Example output:
```
Vector Distance: 0.00028173
```

As your program is calculating the sum, it will also need to print to a file the Euclidean distance of all words that contribute to the sum of the distances. The format for each line in the file is:

```
[ wordN, dist_sqrN ]
```

Once the final sum has been computed write a blank line and the vector distance to the file is the same as it is written to console output. A sample execution is given later in this project specification.

Step VI:
Test your program. Along with the problem specification, you have been provided with 5 files. Two of these files (`sonnets.txt, hamlet.txt`) are by William Shakespeare. Two of the files (`poems.txt, alchemist.txt`) are by Ben Jonson, a contemporary of Shakespeare who has not been strongly accused of writing for William. (Jonson is one of the authors from that era with the most historically documented life.) You should run your program and compare these four files against each other to find out what range of distances to expect between works by different authors and different types of work by the same author (poems vs. plays). Then, run your program on `hamlet.txt` and the final

file, `atlantis.txt` which was written by Bacon. Observe the distance that was returned and, within the context of the previous runs, describe whether you think Bacon wrote Shakespeare's works. Place this short write up (1-2 paragraphs) in a file called `README.txt` (see the next step for details).

Step VII:
Answer the following questions in the file `README.txt`. Note: this write-up constitutes 20% of your grade for this project.

1. Report all pairs of documents you compared and give the vector distance your program reported for the pair.
2. Give your short, but complete, write-up explaining whether you think Bacon wrote for Shakespeare or not. Defend your position. No fancy statistical analysis is required; just keep it fun & simple. This should be 1-2 paragraphs long. Use full sentences and proper English.
3. Did you receive assistance from anyone while working on this project? Who were they and what assistance did they provide?
4. Did you access any other reference material other than those provided by the instructor and the class text? Please list them.
5. How many hours did you spend on this project?
6. What did you enjoy about this assignment? What did you dislike? What could we have done better?
7. Was this project description lacking in any manner? Please be specific.

Step VIII:
For grading, simply submit your updated `word_count.py` file and `README.txt` file.

Programming notes:

1. We will use the **dictionary** container to map a word to its corresponding count. Thus the dictionary will take keys of type string and map them to values of type int. With the map you can use the subscript `[]` operator, using a string key as the index, to set or retrieve the corresponding count.
2. When comparing the word counts, you need to process all the terms in the map. You can do this by using the keys() method for dictionaries. You can then loop over these keys as you would any other list. Note that while you cannot sort a dictionary, you can sort the list of keys before looping over the dictionary.

Sample execution:
Given the supplied files "`common.txt`" (common words), "`sonnets.txt`" (compared) and "`poems.txt`" (compared), here are the first six and the last six lines that should appear in the output file "`comparison.txt`":

```
[ age, 1.09467566915e-06 ]
[ alone, 6.88361137007e-10 ]
[ am, 3.49524039876e-06 ]
[ an, 7.38815140817e-07 ]
[ are, 3.00947917249e-07 ]
[ art, 5.358128774e-06 ]
...
[ why, 1.65241948813e-06 ]
[ world, 1.05945545289e-06 ]
[ would, 6.98267210449e-07 ]
[ yet, 4.73473228795e-08 ]

Vector Distance: 0.0003808030226
```

Further Information:
Wikipedia: overview of the Shakespeare controversy:
http://en.wikipedia.org/wiki/Shakespeare_authorship_question

Project Gutenberg: Source for public domain texts
www.gutenberg.org

Acknowledgement:
This project is based on a project given at the University of Washington, and paraphrased in another project from Spokane Falls Community College.