Part 1 Report

The objective of part 1 is quite simple. We needed to count the words, letters, sentences, and syllables. Using all of these, we then needed to calculate both the Coleman-Liau index and the Flesch-Kincaid score which are both methods that help you calculate what grade level of reading is required to understand the document. Here's how we computed everything:

* Words - We first read everything and then removed all of the periods. After that we used the split function and returned the length of that array.

* Sentences - We used the re library. Using regex, we looked for a ".", '?' or '!'. Following that we looked for a space and a capital letter like most sentences. Then we just took the length of that array.

* Letters - We used a similar method to sentences, but the only difference is we had the regex look for capital and lowercase letters.

* Syllables - This was a more complicated method. There was no clear path for us to move forward in terms of finding a good way to count syllables so we decided to follow the formula given to us in the assignment. Since you gave it to us, we assumed it would be sufficient for the assignment even if it's not the optimal route. Here's how we did it.

1). Count every vowel in a word. If there is only 1 e and it's the last letter, subtract that from the count.

2). We counted all of the dipthongs.

3). After we did that we subtracted the number of dipthongs from the number of vowels and that was the number of syllables (approximate) in a word.

* Coleman Score - Using what we found above, this score is computed using the following formula : (5.88 * (letter_count/word_count)) - (29.6 * (sentence_count/word_count)) - 15.8. There are also protections built in so there are no divide by 0 errors.

* Flesch-Kincaid Score - Just like the Coleman Score, using the variables above, it is computed using the following formula and also has protections for division by 0 :
(5.88 * (letter_count/word_count)) - (29.6 * (sentence_count/word_count)) - 15.8

All of this is built into an Analyze object that automatically computes all of these things on initialization. The main function tries to take in a file as an argument and checks for errors. After that, it creates an analyze object based off of the document you enter and then prints all of it's states. That's all of part 1.

Reflecting on part 1, there were no major difficulties here. There was a small problem working on Python 3 instead of Python 2, but since the program was so simple it was quite easy to change - most of Python 3 is back ported to Python 2, hence the imports we are usings. Since there were no major complex operations, other than maybe the syllable count, we had no issues.

Part1 is pretty thoroughly unit tested, just run the unittests.py file.

Part 2 Report

Part 2 was far more interesting to us than Part 1. We broke up the program into 3 main classes:

1). Category - We used this to create representations of the different document categories, such as business, entertainment etc. Categories initialize themselves with word_counts, and a name, and have another attribute word_scores. This class utilizes the Counter class for the word_counts, which automatically creates key, values based on frequencies - it's basically just a few line shortcut for dictionary creation. Words_scores is calculated in another class, because it requires knowledge of the other categories.

2). TrainingCorpus - This is the class that takes care of training the program based off of the testing documents provided. This is where we used the formulas provided in the assignment description to give word's scores for each of the categories provided.

On initialization, it takes a list of categories and updates each Category word_score attribute.

It's first attribute is total_word_count, a Counter of all the words in all the Category's combined. This makes doing some calculations easier.

It's second attribute is category_list, a list of Category objects that we created during training.

Basically, we begin with a Counter to find the frequency of each word in each Category. Using these dictionaries, we can calculate the individual word score for each category using the formula:

log(count of word in document) - log(count of word in every other document)

We also have some protections from log(0) built in to make sure there are no errors.

3). Document - This is the class we use to represent our testing documents. It stores it's words_list, category_scores, and informative_words as attributes.

We can then analyze a current document and try to place it in a category. Here we just take all of the words out of the document and then update the score for each associated category based on that word. Since we created a score for each word by training, we can go through each word and update the score from each word in the Document's categories. Whichever category has the highest score is the category the document most likely belongs to.

In addition, while we calculate the document score, we keep track of the most informative words. Basically, if the word score we encountered was higher (which means it's important for classifying) than the minimum word score in our top 5 most informative words, we just replaced the minimum with the new score. That way we could return the 5 most informative words at the end.

After the class construction, we have two 'main' functions, main() and main2(). The first main function runs the search against the smaller dataset, where we are given a business file. This function prints out relevant stats, like the category total scores and the most informative words.

The main2() function is more complex. We manually separated 100 documents from each category into a folder called ‘bbc-test’, and left the rest in ‘bcc-train’. We then used the files in bbc-train as the training documents for each category. We then ran bbc- test documents against this data set, and print out the results.

Many of the categories match incredibly well, specifically the categories Tech, Politics, and Business have 95%+ accuracy. Sports follows up with 74% accuracy, and lastly entertainment with only 55% accuracy.

One hypothesis is that entertainment discusses a wide range of things: sports-entertainment, political-entertainment, tech-entertainment - there’s a lot of cross over with other categories. Sports was mistaken with politics 20% of the time, suggesting that the focus on writing about people might cause the confusion.

This part produced some difficulties for us unlike part 1. For starters, we had some problems collaborating between our machines which was actually the biggest issue. Ben worked in Python 2 and James worked in Python 3 initially so this caused lots of issues for us closer to the submission point.

Also, with our equations we had some log problems as we didn't want to produce any log(0) errors. Since it's been a while since either of us took a math course, it took us a minute to catch.

Reflecting back on this assignment, our lives would have probably been easier if we spent more time developing in a clean way rather than going back later and fixing it.

Planning out a strong OOP approach is one of the most important lessons learned from this assignment.

Part 1 wasn't really a problem given it's simplicity, but Part 2 became sort of a mess that took a while to clean up. Ultimately though, this was quite a fun assignment and we look forward to doing the rest of the search engine.