COM3110/4115/6115: Lab Class

Computing Word Overlap

1. This week's lab provides practice in word counting over multiple files. More specifically, we look at computing the level of *word overlap* between documents as an indicator of document similarity, which might arise in circumstances such as document duplication, text reuse (e.g. plagiarism), even simply having the same topic.
2. Download the file `NEWS.zip` from the module homepage (see: `Lab Classes > Data/code files`). Unzip this to get a folder `NEWS` of short news articles (with names `news01.txt`, `news02.txt`, *etc*). Also download files `compare_STARTER_CODE.py` and `stop_list.txt`, and place them *alongside* the unzipped news folder. A *stop-list* is a list of 'unimportant' words — words that are often ignored during word counting (discussed later in the course).
3. Rename the *starter* code file as `compare.py`, and extend it to complete the lab exercises. The script might be called from the command line as follows:

   ```
   > python compare.py -s stop_list.txt NEWS/news01.txt NEWS/news02.txt
   ```

   Here, flag `-s` identifies `stop_list.txt` as an (optional) stoplist. The subsequent command line arguments (`news01.txt` and `news02.txt`) identify the files to be compared. The script can alternatively be called as:

   ```
   > python compare.py -s stop_list.txt -I 'NEWS/news??.txt'
   ```

   Here the `-I` option indicates the *input files* by using a *pattern*, which here matches all files in the `NEWS` subfolder, with name `news??.txt`, where `?` can be *any character*. The starter code assigns the list of names for the input files to the variable `filenames`.

   The starter code also loads the stoplist, storing its words in a *set*, in the variable `stops`. A set is a good choice of data structure here, as it allows fast, hash-based look-up, via a simple test such as "`w in stops`". If the optional stoplist is not specified, `stops` will store an empty set, so, in fact, the same test can be used, but will always return `False`.

   Study the starter code provided, to make sure you understand it.
4. Starting with the case where there are only *two input files*, extend the code to compute and print a *word overlap score* (explained below in 7.) between the two texts, e.g. as in:

   ```
   > python compare.py -s stop_list.txt NEWS/news01.txt NEWS/news02.txt
   NEWS/news01.txt <> NEWS/news02.txt = 0.0187
   ```

   Some suggestions about components to implement as part of this script follow.
5. TOKENISATION: To simplify, you might treat the words as being all the maximal alphabetic sequences found in the file (which can readily be extracted from each line with a regex findall). To improve overlap, you should convert the text to lowercase, so that different case variants are conflated in counting.
6. COUNTING: You might define a function (e.g. `count_words`) which counts the occurrences of (non-stoplist) words in a file, and returns these counts as a dictionary, i.e. with words as keys, and counts as values. (Alternatively, you might define a simple class, whose instances store this information, and have the word counting function as a method.)

7. COMPARISON: You might define a function which, given the counts for two files, computes a measure of their word overlap. We will use versions of the *jaccard coefficient*. A simple version ignores the counts of words in the files, and instead treats each as just a *set* of word types. For word sets A and B, this measure is computed as:

$$\frac{|A \cap B|}{|A \cup B|}$$

8. For a count-sensitive version of a jaccard metric, we might use the following. For compared files A and B, let w_A and w_B denote the counts of word w in the two files respectively. Our measure is then computed as follows (where $\Sigma_{w \in A \cup B}$ here ranges over the full set of terms appearing in either file):

$$\frac{\Sigma_{w \in A \cup B} \texttt{min}(w_A, w_B)}{\Sigma_{w \in A \cup B} \texttt{max}(w_A, w_B)}$$

9. Having produced a script that can compare two files, you might modify it to perform pairwise comparison across more than two files, e.g. as in the following, where the pattern picks out three input files for comparison:

```
> python compare.py -s stop_list.txt -I 'NEWS/news0[123].txt'
NEWS/news01.txt <> NEWS/news02.txt = 0.0187
NEWS/news01.txt <> NEWS/news03.txt = 0.0118
NEWS/news02.txt <> NEWS/news03.txt = 0.0112
```

   Given the number of comparisons

   made here, you might modify your code to store up the scores, and then print out only the top N scoring comparisons.

10. Apply your code to the set of news articles to see if you can find the following cases of related files:

    - *duplication*: two of the news files are identical, in terms of their textual content (although they are *not literally identical*, as might be tested using unix `diff`)
    - *plagiarism*: one of the news files has been produced by cutting and pasting together text from two of the other files
    - *related topics*: three of the articles address the same news story, as given by three different newspapers. Do these separately authored presentations of the same story exhibit higher overlap than articles on unrelated topics? (Note, two of these related articles appear within the first 5 news stories.)

11. You can investigate whether similarity detection works better with the simple or the weighted metric, and whether the use of a stoplist helps or not.
12. If you have time, look back at your code and consider how the functionality might be grouped into classes, to make it easier to understand (and potentially more reusable).