COM6115: 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), or even simply having the same topic.

2. Download the code/data file (zip) for this lab from Blackboard. Unzipping it gives a folder with files `compare_STARTER_CODE.py` and `stop_list.txt`. (A *stop-list* is a list of 'unimportant' words – which are often ignored during word counting.) It also contains a subfolder `NEWS` of short news articles (with names `news01.txt`, `news02.txt`, *etc*).

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. These file names are stored as a list in the variable `filenames`.

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

   Later, we will want to call the code to take *all the files* in the NEWS folder as input, but the way to do this depends on the operating system (OS) being used.

   UNIX/LINUX/MAC: simply use 'wildcards' in the filename, as in the following example:

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

   WINDOWS: the above convenience is not available in the Windows terminal, so the code instead implements it via the `-I` option, which must be supplied with a *pattern* to match the file names, as in the following example:

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

4. Study the starter code provided, to make sure you understand it. When completed, this code takes the files listed on the command line, and performs a *pair-wise comparison*, printing out the multiple pairings and their similarity scores. You will see that there are some functions with 'dummy' definitions, which you will need to fill out.

5. First, there is a 'dummy' definition for a function `count_words`. This takes a file name and a 'stoplist' as inputs, and returns a dictionary of counts of the words in the document. Counting should ignore any words in the 'stoplist' set.

   Regarding tokenisation, 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 simple regex, and the regex `findall` method). To improve overlap, you should convert the text to lowercase, so that different case variants are conflated in counting.

The script overall has a command line option "-p", to switch on Porter stemming. A function stem_word is defined, to stem individual words. Your code should apply stemming if the "-p" option is present. NOTE: the stoplist has *not* been stemmed, so words should be checked against the stoplist *before* stemming is applied.

6. Secondly, there is a 'dummy' definition for a function jaccard, which you should complete. This computes a similarity score for two documents, based on their word-count dictionaries. A command line option "-b" determines the calculation that is done. If it is present, a *count-insensitive* metric is applied (giving a form of so-called *binary term weighting*). This simple metric 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:

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

If option "-b" is *absent*, then a count-*sensitive* version of a jaccard metric should be computed. In comparing files A and B, let w_A and w_B denote the counts of word w in the two files respectively. The metric is then computed as below (where $\Sigma_{w \in A \cup B}$ here ranges over the full set of terms appearing in *either* file):

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

7. Having completed the function definitions specified above, you can call it to compare multiple files identified by an appropriate expression (as indicated in part (3) above). Thus, `NEWS/news0[123].txt` picks out files `01..03`, whilst `NEWS/news0?.txt` gets files `01..09`, and `NEWS/news??.txt` gets *all* files.

   The number of comparisons being computed/printed is quadratic in the number of files selected, rapidly making it hard to inspect them. Hence, you should modify the code to print out only the top N (e.g. 20) scoring comparisons. The similarity scores for the multiple pairings are stored in the dictionary results. For this task, you will need to sort this dictionary based on its *values*, rather than its *keys*.

8. Apply your code to the *full* 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.)

9. You can investigate whether similarity detection works better with the simple or the weighted metric, and whether the use of a stoplist and/or Porter stemming helps or not.