Locations and other general information

- Scripts located in /home/students/kwiseth/lab*n*
- None of the scripts requires any command-line options. All data elements are accessed by the script as it executes from the directory specified.
- Previous versions of scripts have been moved into a /sandbox sub-directory within each /lab*n* directory, so hopefully won't be difficult to find the script.

Lab 2: POS tagging using HMM

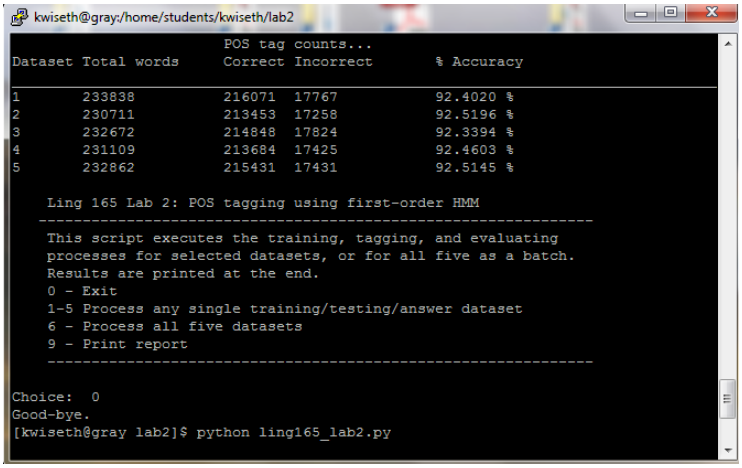
Path/script: `/home/students/kwiseth/lab2/ling165_lab2.py`

Menu at startup: Displays options to process individual datasets or all 5 datasets, and to print the report.

Output: Displays status messages during processing and displays current report at the end of the process.

Saved artifacts: Saves the `afd_n.save`, `bfd_n.save` pickled dictionary file from training process and the tagged output file from test data (`tagged_output_n.save`) for each dataset when processed, where *n* is the number of set of files from the prepared 5-fold cross validation method dataset. Also saves the pickled dictionary of the results-report (`lab2_report`).

Sample run:



I've left the final test run results in the `\lab4` sub-directory, so if you select 9 from the menu you'll see the last results report.

Lab 3: Naïve Bayes classifier and word-sense disambiguation

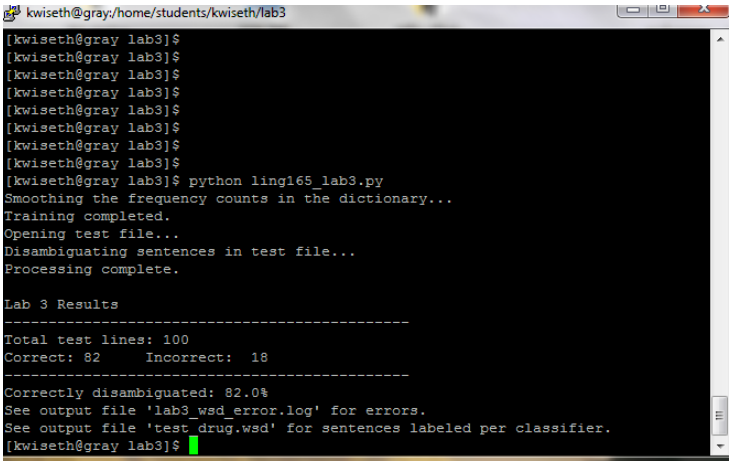
Path/script: `/home/students/kwiseth/lab3/ling165_lab3.py`

Menu at startup: None.

Output: Displays report of percentage sentences correctly disambiguated—**82%** with the given training and test data.

Saved artifacts: `test_drug.wsd`, `lab3_wsd_error.log`

Sample run:



For Lab 3, I think we're to understand that 'words' are 'features' and that P(w | c) is just another way of saying P(f_j | c), correct?

Lab 4: Content selection for single document summarization

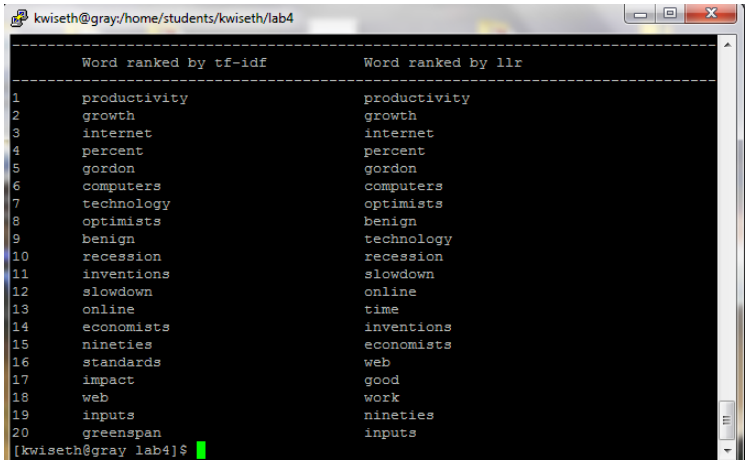
Path/script: `/home/students/kwiseth/lab4/ling165_lab4.py`

Menu at startup? No. [However, note that the `newyorker.txt` file has been processed in advance using Hahn's `clean.py` script, with its output saved to '`newyawker.txt`' file, and that '`newyawker.txt`' must be available to ling165_lab4.py at runtime.]

Output: Script displays status messages during processing and then displays a two-column report of top-twenty words (per each approach) in descending order, starting from most informative.

Saved artifacts: None.

Sample run:

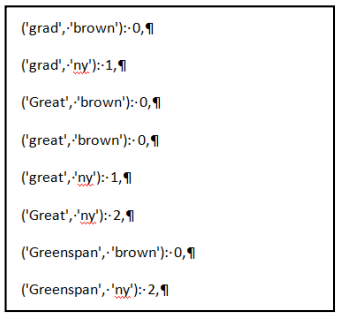


For Lab 4, I ran into a couple of issues, starting with differences between character sets (utf-8 and ascii), which may have had to do with the settings on putty, but during the course of developing and testing (and looking at output), I kept ending up with the codepoints ('\xe2\x80\x94', for example for em-dash) in some of my results. As a workaround, I pre-process the 'newyawker.txt' file (the New Yorker article as output from your `clean.py` script) and convert these utf-8 codepoints to their ASCII character equivalents. For the most part, these are various punctuation marks which may be

eliminated anyway, but during testing I wanted to be able to see what was getting collected (or not) without looking at '\xe-etcetera's stuff, so this was mainly for cosmetic purposes.

Another issue was the necessity of eliminating stop-words before collecting word (term) counts, or not. I would have thought that the frequency of any of the various stop-words, particularly words like 'the' and 'of' and 'to' etc would have been off-set (and hence, such words would not show up in the top-20) because these words would appear with the same relatively high frequency when compared across all the brown documents—but this didn't seem to be the case insofar as the llr list was concerned, which leads me to wonder if I'm still doing something wrong in Lab 4's code. In the end, to obtain a list of top-20 informative words per llr, I by-pass stop-words before collecting counts and calculating tf-idf and llr. For performance purposes, it's probably a good idea anyway.

I tried to normalize the text as much as possible before collecting term-frequency counts, but this still isn't perfect. I would have liked to have lower-cased just the initial word in a sentence but keep upper-casing alone elsewhere in the article, and I thought it might have been good to keep certain words together. After examining an interim version's dictionary results and seeing that different counts were being accumulated (as shown in this screenshot), I realized it best to simply lower-case everything before counting.



Since we're comparing to documents from brown, and since the New Yorker copy-edit style is (in some cases) different from other news/magazines, two changes I made to the text were putting "per" and "cent" back together as a single word, and also hyphenating the string "output-per-hour," since both of these might be found in the brown documents. (If our "corpus for comparison" were 500 New Yorker articles rather than Brown documents, this is perhaps a moot point.)

Finally, I also eliminated numbers, including years. In previous iterations of the scripts (before doing so), the years '2004' and '2000' show up in the top-20 tf-idf list. Since the New Yorker article is being compared to 500 news documents from brown, and since the Brown corpus was initially created in the 1960s (with I think an update in the 1970s?), I'm guessing that these dates would rightly show up as 'informative' since they wouldn't be mentioned in our brown data. So in the end, I just decided to by-pass numbers and dates.

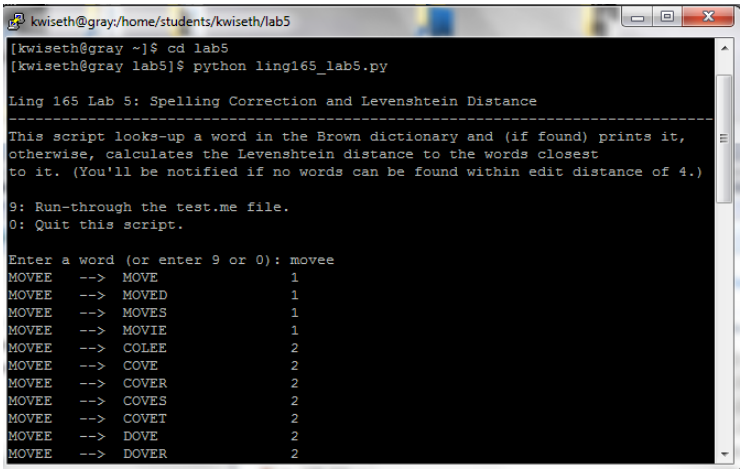
All these decisions may have led to an incorrect handling of the problem, but in looking at the lists generated, it seems that these words are okay. I don't like that the llr list is so close to that of the tf-idf—makes me think that something's wrong--but hopefully, it's okay.

Lab 5: Spelling correction (using Levenshtein, aka minimal edit distance)

This project doesn't actually 'spell correct' but rather offers possible corrections to the user.

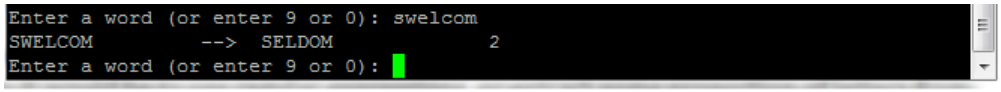
Path/script:	`/home/students/kwiseth/lab5/ling165_lab5.py`
Menu at startup?	Yes. Enables entry of a single word or processing the 'test.me' file.
Output:	Displays list of words from Brown sorted by Levenshtein distance, limited to edit distance (Levenshtein distance) of 1-2. If no words within 1-2, then prints out words found within Levenshtein 3-4.
Saved artifacts:	None.

Sample run:



Depending on the word entered, this script may run VERY slowly. The 'minimumEditDistance' function in this script programmatically replicates the pencil-and-paper approach we used in class to determine the edit distance between a source and a target word. The 'source' word is the word entered by the user, and the 'target' word is a word from the brown dictionary file.

The brown dictionary words passed to the minimumEditDistance function are from a pre-selected subset. I tried a couple of different approaches to minimize the brown search space two of which remain in my script: `get_brown_ltd_narrow(some_word)` and `get_brown_ltd(some_word)`. The more limiting of the two functions (`get_brown_ltd_narrow`) provides results more quickly but it's flawed in that it's too limiting: if the user had been trying to type "welcome," that choice won't be found, for example:



However, using `get_brown_ltd (`instead of `get_brown_ltd_narrow`) increases the processing time significantly—so much so, I thought the program had hung-up. I'd like to figure out a better way of limiting the search space without losing content, so perhaps the better way to do it would be to use regular expressions (although I think regex take longer than using string functions in python?) or to have the pre-selection IF statement also include one or two characters from the middle of the word entered, so that only words from brown will be selected for search space if they begin, end, or have one (or some percentage) of characters in common.

From the subset, each potential target word is sent to the minimumEditDistance function (with the source word), and the function calculates and returns the Levenshtein distance. The results are collected into one of two lists according to Levenshtein distance—one list contains words within a distance of 1-2, the other contains words within 3-4. Finally, the script displays the results of one list as appropriate for the word entered. I'm thinking that for most practical purposes, the user would be looking for a word within a range of 1 to 4 edits, but would prefer to see only those words within 1-2 if they exist.

Of course, none of this happens if the word entered is found in the brown dictionary—the word is displayed accordingly.