Python Assignment – Yujun Liu 2494841

Generating the list of abbreviations will consist of reading in the input file, computing the abbreviations, filtering out non-unique abbreviations, and then selecting the lowest-scoring abbreviation and writing the output file. Additionally, as specified in the assignment, the entire code should be able to be run from the command line, necessitating an if \_\_name\_\_ == ‘\_\_main\_\_’ section.

In preparation for computing the abbreviations, for which the scores are calculated based off the letters only and are affected by the structure of the words in the name, it is necessary to format each name into words free from non-alphabetical characters. This is easily achieved with regex’s split method, which, if applied to the name after apostrophes have been replaced with empty strings, can return a list of words for each name, split appropriately where there once were non-alphabetical characters. A copy of the original names is also kept to not have to get them again when writing the output file. This will be performed in section 1 of the python code.

Computing the abbreviations comes next, but first requires associating each letter with a score with special cases for if the letter is in certain positions in a word or if it is the letter e. Given that the score is calculated from the last two letters of the abbreviation, and that the letters come from the words in the name, it would likely make things easier to tie each letter within the name to a score before the abbreviation is computed, or in other words, to store the letters and scores in the same format. This can be done by reading in the values.txt file to a dictionary and creating a nested list for each name with the appropriate letter scores for each letter in the sub-list corresponding to each word, which allows the indexing to access a specific letter to be the same as the indexing to access the score of that letter. This makes easier accessing the scores for further parts of the program as well as the modification of existing scores for position or the letter e. To compute the abbreviations, the code will be like that of the subseq3 function from the labs but given that the first letter of the abbreviation is fixed, the subseq2 function is more appropriate and will be applied to every letter but the first in a name. Furthermore, subseq2 can be modified to take in both lists of integer letter scores and strings, so that while the order of the letters in the string will be maintained when abbreviations for the second and third letters are formed, the score for each abbreviation can also be calculated while following the same ordering rules as the letters. The tasks detailed here will be performed in section 2 of the python code.

After the abbreviations and their scores are obtained, two layers of filtering need to be performed. First would be filtering out duplicates within each name, which can be accomplished by using a dictionary to keep track of the minimum score for each abbreviation of a given name and then setting the abbreviations and scores to the updated dictionary keys and values. The second layer of filtering consists of discarding abbreviations that correspond to more than one name and can be accomplished with sets. This time, a set will keep track of which abbreviations have already showed up, and if they show up again, add them to a second set. After going through every abbreviation of every name, goes through them again to remove any abbreviation from the second set. Duplicates here can now only come from other names since they were removed within each name earlier, so no within-duplicated abbreviations are being discarded. These two filtering steps will be performed in section 3 of the python code.

Now, having two within-and-across-deduplicated nested lists, one of abbreviations and the other of their scores, selecting the lowest scoring abbreviation(s) for a given name can be done with a for loop and a variable keeping track of the lowest score and selecting the abbreviations with that score on a second loop-through. The original names were kept from the first step, which will save effort when writing to the output file. When writing the output file, the original name is printed and then the abbreviations, with a new line instead if there are no valid abbreviations. This will be performed in section 4 of the python code.

Lastly, a main() function will run the other parts of the code from start to finish. As specified in the assignment, it takes in the name of the input text file without the .txt extension. This main() function will be written in section 5 of the python code, and the user interface portion that allows calling this main() function from the command line in section 6.

While the code was being written, testing was performed to ensure that each step, and by extension the entire program was doing the intended job. As a matter of practicality, a reduced set of the trees.txt file was used as the test file, containing only the first five names: ‘Alder’, ‘Crab Apple’, ‘Common Ash’, ‘Silver Birch’, and ‘Downy Birch’, plus an additional “Yujun’s Test1ng\_Elm” to test that non alphanumeric characters were being removed and ‘Aldre’ to test that duplicate abbreviations across names were being removed. The code in section 1, in additional to the original names, returns [['Alder'], ['Crab', 'Apple'], ['Common', 'Ash'], ['Silver', 'Birch'], ['Downy', 'Birch'], ['Yujuns', 'Test', 'ng', 'Elm'], ['Aldre']], indicating that the apostrophes were removed correctly, and the words were split up correctly. Running section 2 returns [[[0, 16, 11, 38, 5]], [[0, 16, 27, 5], [0, 9, 10, 18, 20]], [[0, 21, 10, 11, 23, 5], [0, 16, 5]], … ]. It appears that the letter scores corresponding to ‘apple’ ends with a 20, which is correct for words ending with e. The positional score modifiers also seem to be correct for the word “common” immediately afterward: c: 0 o: 20+1 m:8+2 m:8+3 o:20+3, n:5. This results in the abbreviation function outputting ([['ALD', 'ALE', 'ALR', … ], … ], [[27, 54, 21, …] …]]), which seems correct as ‘LD’ should give 16+11, ‘LE’:16+38, and ‘LR’:16+5. The next step is deduplication within, and it is most obvious that ‘Common Ash’ ‘s ‘COM’ abbreviation would appear twice and have scores 31 or 32. Running the deduplicate within function outputs ([ … , ['COM', 'COO', 'CON', …], … ], [ … , [31, 44, 26, …], …]) and demonstrates that the abbreviation is only being kept once and that the lowest score is being used. As for deduplicating across, ‘Alder’ and ‘Aldre’ should both have the ‘ALD’ abbreviation. In the results of the deduplicate across function, ([['AER'], … , ['ARE']], [ … ]]), one can see that the ‘ALD’ abbreviation has been removed, indicating that the deduplicating across function worked correctly. Lastly for testing what would not be immediately evident in the output file is the function that selects the best abbreviations. It outputs [['AER'], ['CBA'], ['CNA', 'CAH'], ['SRB', 'SBH'], ['DYB', 'DBH'], ['YNE', 'YTE', 'YTN'], ['ARE']], which suggests that ‘Common Ash’ has two abbreviations that share the lowest score. Expanding the results of the previous function that deduplicated across, ([ …, ['COM', 'COO', 'CON', 'COA', 'COS', 'COH', 'CMM', 'CMO', 'CMN', 'CMA', 'CMS', 'CMH', 'CNA', 'CNS', 'CNH', 'CAS', 'CAH', 'CSH'] , … ], [ … , [31, 44, 26, 21, 37, 26, 21, 33, 15, 10, 26, 15, 5, 21, 10, 16, 5, 21], … ]), one can see that this function is correctly selecting the lowest-scoring abbreviations.