Mahesh Kumar Ravindranathan
CSCI 5832
Deterministic Segmentation

Part 1:

The source of Failures:

- The frequently used words are grouped together and are added as part of the dictionary (Example: is & a, isa is part of the dictionary)
- Frequent words are also added as affix to the normal words (Example: is & real, isreal is part of the dictionary)
- Some words don't exist in the dictionary (Example: cuboulder, iphone6s)
- Words with no meanings are present in the dictionary (Example: tth, ntl, atl)
- There are some words which are short form of the original words but are not contained in the dictionary (Example: saga (Screen Actors Guild Award) which is not present in the dictionary)
- There are some words which begin with the frequently used words like "the" or "my" or "I" (Example: "myth" has the word my)
- There are some words which are plural and has the extra "s" or some other form in the dictionary but training set expects only the stem (Example: "show" is expected, "shows" exist in the dictionary)
- There are multiple frequently used words starting with the same alphabets (Example "the" and "there")
- Since Maxmatch is a greedy algorithm, it tries to make the first word as big as possible even though the second word being bigger would make more sense (Example: "totalladega" becomes "total lad ega")

There are are around 4 kinds of frequent failures which includes: Words with no meanings in the dictionary, Greedy Algorithm Failure, Words having the frequently used words as prefix and present in dictionary and frequent used words attached together present in dictionary.

Part 2:

WER (Word Error Rate) on the Hashtag Training Set: 0.661224489796

Part 3:

There is a module called maxMatch which implements the default max-match algorithm. There is module called computeAverageWordErrorRate which implements the calculation of WER. There are multiple modules that work in sync to generate the improved segmenter system (intermediateOptimization, maxMatchOptimized and intermediateOptimization)

Approach chosen to get maximum accuracy and reduce the Word Error Rate:

Initially I spent almost a day or two, removing words from the dictionary using shell/perl/awk/sed etc.. Removing words did not give me good Word Error Rate. There is no way that the dictionary can be fixed completely automatically. I thought it was better to filter via python in real time than to filter out since I can't show the words that were removed or were absent in my dictionary during submission. I then tried min-match where I called the min match function recursively from max-match. It did not give better Word Error Rate as well. I then thought about implementing only input optimization function. Error rate reduced. I thought what if I can call an intermediate optimization as well. It helped me to get the error rate even lesser. In the end, there were only few errors and I thought I could make error rate zero by making some optimization at the output. I realized that there were certain scenarios which may occur and the code can fix them automatically even at this point. I then implemented output optimization function.

These are the Optimizations that I have implemented.

1) Avoid words from being added to the dictionary among the 75,000 words
    a. Don't add words into dictionary that starts with "the" and is followed by any of the following alphabets: "bcdeghjklpqtuvwxz"
    b. For the top 75 frequently used words, make a list of words which have the remaining 74 words attached to it as prefix. (Ignoring "without")
    c. After the top 75 words, if the length of the word is greater than 4 and the last alphabet is a vowel, I am not adding such words to the dictionary.
    d. Removing a list of words which makes no sense and reduces the WER (Around 10-15 words)
    e. After the first 100 frequently used words, I don't allow an entry of two alphabet word to the dictionary which contains both the alphabets as vowels.
    f. After the first 500 frequently used words, I don't allow an entry of word starting with the first 500 frequently used words followed by an alphabet.

2) Before passing to the max-match algorithm (Greedy Algorithm), I pass it to a function called optimizedInput. This function checks if the word begins with frequently used words of the dictionary. If the word does being with a frequently used word, then it does a normal max-match to figure out the length of the first word in the case of greedy algorithm. If the length is lesser than a threshold, I assume that Greedy Algorithm was greedy and did not consider the first word to be a frequently used word. In such a case I consider the first word to be the frequently used word and pass the remaining content to customized max-match algorithm if the remaining content as a whole word is not present in the dictionary. If the remaining content is also present in the dictionary, then I just return those two words as final output.

3) In the Optimized max-match algorithm, after the words are formed by greedy approach, I consider the length of the word formed. If the length is too big or small I assume that the word was correct after all, if the length is not too big nor small then I send it to intermediateOptimization function which tries to determine if the greedy algorithm was too greedy or not.
4) In the intermediateOptimization function, I check if the word contains the frequently used words. If it does start with frequently used words, then I check if the remaining alphabets as a whole are present in the dictionary or not. If yes, I return them as a list. If that is false, then I check if the word without the frequent used word makes sense or not, if yes then I return a list again of frequently used word and the other word (without the remainder). If both of the above cases are not true then I return the input as it is.
5) In the optimizedOutput, I wrote few conditions to optimize output
    a. Generally, the last word is never a 1 or 2 alphabet word. Hence if it is a one or two alphabet word, I add it to the previous word and check if it is part of the dictionary. If it is not part of the dictionary, then I add it to the previous word (assuming that word is present in the dictionary and it makes sense.)
        i. The number of words is greater than three in the input sentence.
        ii. The number of words is equal two in the input sentence.
    b. Generally, the last word contains 3 alphabets only once in a while. Hence if it 3 alphabet word, I check if it is one of the frequently used words. If yes, I keep it as it is or else I follow the same method as above.
    c. Due to wrong words in dictionary, in spite of removing some words, there is a possibility of a wrong word present in dictionary. I try to fix the issue by checking for 3 alphabet words which are not frequent and I divide and assign them to the adjoining words if they exist in dictionary.

I could bring the WER to 0 for the test set by coding based on multiple conditions but I think in general, the WER that I got after making the code to be optimized in general was around:
`0.0744047619048`