ASSIGNMENT 1 PART 1

AKASH REDDY A (EE17B001)
NAYAN N (AE17B010)

Question 1

A really simple top-down approach to sentence segmentation of English texts is to use the period character and question mark character to divide a long string of characters into component sentences. This is a top-down approach as we are using our prior knowledge that sentences in the English language usually terminate with a period or question mark hence making it a good candidate to recognize sentence boundaries.

Question 2

The approach suggested above would do a good job in most cases but it would fail in some very obvious situations. For example, when the sentence contains an abbreviation such as Mr./Mrs./St. it would falsely recognize the usage of period here as a sentence boundary. A simple way to circumvent this issue would be to hardcode some rules into the segmentation script ensuring that the most common occurrences of the period character used outside of the context of a sentence boundary are ignored.

Question 3

NLTK's punkt sentence tokenizer has a bunch of predefined rules to improve the quality of sentence punctuation. It has a pretrained model recognizes sentence boundaries using period and question marks but has rules to ignore sequences like initials, abbreviations and other non boundary uses of period characters. It also allows us to train a model with custom rules given that we have a large enough corpus. It improves upon the top down approach as spoken about in question 1 using the general methodology spoken about in question 2 in a more formal and standardized way.

Question 4

CODE SUBMITTED.

Output of segmented sentences is written into "output/segmented_docs.txt" and "output/segmented_queries.txt"

The written output changes accordingly for simple top down approach and for nltk punkt tokenizer approach, based on the "-segmenter" flag provided while running main.py.

```
Execution time of top-down approach : 0.00989532470703125 seconds
Execution time of nltk punkt tokenizer : 0.3748948574066162 seconds
```

a. The first method (i.e. top down approach performs better when speed is our criteria of evaluation. It is about 40 times faster than nltk punkt sentence tokenizer. If we have a very basic requirement where we are certain that border cases will not cause an issue in tokenization we can use a simple rule based top-down approach for better efficiency.
b. The nltk punkt sentence tokenizer is much more accurate than the simple rule based top-down approach as it has fewer false positives for sentence boundaries.That is the nltk's sentence punctuator is able to recognize when the period is used to specify a sentence boundary and when it used in a different context like Mr. , Dr. , St. etc.

Question 5
The simplest top down approach for word tokenization would be to split words by whitespace characters (specifically space ‘ ‘) to get a list of words from a sentence. We can extend this slightly to accommodate badly formatted sentences where a space character is not used after a comma ‘,’ character is also recognized as a word boundary. This is a top-down approach as we are using our prior knowledge about what generally classifies a word boundary, i.e. a space character.

Question 6
NLTK’s Penn Treebank tokenizer is a bottom-up approach as the sentences that are passed to the tokenizer are split into words using regular expressions based on the Penn Treebank corpus. Although the methodology for splitting sentences into words are condensed into a bunch of rules, these rules were derived by observing patterns in an extensive corpus.

Question 7

CODE SUBMITTED.
The word-tokenized output is written into output/tokenized_docs.txt and output/tokenized_queries.txt.

The naive or Penn Treebank tokenizer can be chosen by giving the “-tokenizer” flag as “naive” or “ptb” (while running main.py), and the written output in the aforementioned .txt files changes accordingly.

Recording the time taken to tokenize the Cranfield **query** sentences into words:

```
Simple top-down Tokenizer:  0.0006031990051269531 seconds
Penn Treebank Tokenizer: 0.016706466674804688 seconds
```

The simple approach is clearly faster than the Penn Treebank tokenizer (around 28 times faster) for the same task.

(a) In cases where the text is plain and has mostly whitespaces, commas, and periods through the sentences, the simple approach does as well as the Penn Treebank tokenizer in terms of accuracy. However, it is much faster because there are only a handful of rules and delimiters to apply during tokenization. Therefore, the simple approach does better overall.
Example: The quick, brown fox jumps over the lazy dog.
(b) In sentences which are fancily decorated with punctuation and contractions, the Penn Treebank tokenizer accounts for almost all such delimiters and does a more accurate job of tokenizing the sentence. The simple method would still run much faster, however the results would be too inaccurate to be acceptable. Therefore, the Penn Treebank tokenizer does a better job.

Example: “Why would he?!” he half-exclaimed, half-wondered. “Why’d he steal your $200?!”

Question 8
While the goal of both stemming and lemmatization is to reduce all inflectional forms or derivatives to a common base word, they are different in approach.

- In stemming, we prune the endings of words, hoping to remove suffixes and arrive at the correct base word most of the time. This approach is computationally faster, as it involves following a fixed algorithm depending on the suffixes a word has. Stemming results in a base word that is not necessarily a dictionary word. Hence, stemming more commonly collapses all derivationally related words of a given word, even if the parts-of-speech are different for instance.

In information retrieval tasks for example, there tends to be lower precision but higher recall, as multiple words can get merged into the same stem, leading to overall more matches. These may add a few more relevant matches (higher recall), but many of them can also tend to be unrelated, leading to lower precision in general.

- On the other hand, lemmatization involves the use of a corpus of words. It uses part-of-speech information and morphological analysis to obtain the “lemma” or base word from within the corpus. Since lemmatization involves scanning the corpus and processing extra information, it consumes more time and computational power, but always returns a dictionary word as the lemma. Due to this, lemmatization commonly only collapses the different inflectional forms of a lemma while more rigidly separating derivationally related words and different parts of speech.

Lemmatization tends to result in more precision than stemming in an information retrieval context, as it only accepts matches that contain the precise dictionary lemma and context. However, it gives lesser recall than stemming in general, because stemming leads to more results being accepted as matches (of which some would be relevant).

Question 9
Search engines require the following functionality:
- Most people look for the most relevant results, and usually do not go past the first few pages of results. Here, precision matters.
- The web usually has many pages relating to most popular topics, and this means that it is not rare to have related results. Therefore, there is not much need to do an exhaustive search with high recall which also allows unrelated results to pass through.
- People are willing to skim through a few unrelated pages, but are most likely going to be displeased if most results that they see are not what they want.

In conclusion, precision matters more than recall in this application. **Lemmatization** figures out the true dictionary lemmas of the words, thereby reducing the noise in the results and pushing up the most relevant results to the first pages for the user’s convenience. In stemming, on the other hand, faster processing of the queries and documents occurs; but it is possible that

multiple words have the same stem, and this can lead to noisy results. Also, when all derivationally related words (not just inflectional forms) are clubbed, we could obtain a few more related results, but the contextual information may be lost leading to many irrelevant results as well.

Therefore, generally lemmatization is better in the search engine context, but it does come down to pragmatics and specific application.

Question 10
CODE SUBMITTED.
The lemmatized output is written into output/reduced_docs.txt and output/reduced_queries.txt

Question 11
CODE SUBMITTED.
The stopword-removed output is written into output/stopword_removed_docs.txt and output/stopword_removed_queries.txt

Question 12
Instead of providing the program with a list of stopwords (top-down approach), stopwords could be figured out from the given domain information/corpus using certain metrics in a bottom-up fashion. Stopwords are those words that exist across virtually all sentences/documents, therefore carrying negligible power of discriminating or representing the sentence/document.

Thus, a metric such as document frequency can be used to determine if a word is used frequently across all documents, making it likely to be a stopword. If we use the TF-IDF metric, terms with very low TF-IDF scores are most likely to be stop words. The “Inverse Document Frequency” component of TF-IDF will bring down the score drastically if the word is frequently repeated across most documents and has low discriminating power.

Using such a bottom-up approach could also help us figure out domain-specific stopwords, which would otherwise be ignored in a generic top-down list of stopwords.

****

References Used:

1. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
2. https://www.nltk.org/_modules/nltk/tokenize/treebank.html
3. https://www.nltk.org/_modules/nltk/tokenize/punkt.html
4. Prof Sutanu Chakraborti’s CS6370 Lectures
5. https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/
6. http://www.nltk.org/book/ch05.html
7. https://stackoverflow.com