Assignment 1 - Part 1

CS6370: Natural Language Processing

Shreya Nema: BS17B033
Ayush Maniar: EE17B041

1. Sentence Segmentation using delimiters:

Simplest and top-down approach to sentence segmentation for English text would be to use sentence delimiters. These delimiters in English are period(.), the exclamation mark(!), question mark(?) that marks the end of a sentence. The text encountered between these delimiters can be stored as a sentence.

2. Would it work?

The sentence segmentation using delimiters won't always work especially in the case of sentences with abbreviations. For example, "His roll no. is not correct in the documents." will split the text into two sentences :
["His roll no", "is not correct in the documents"]
Here a single sentence got split into two arbitrary sentences and the context of the sentence is lost. Another approach could be to split the text into sentences whenever a period and a space followed by capital letters is encountered. Even this top-down approach won't work in the following case:
"Mr. Smith and Mr. Jacob are best friends", here the text will split into 3 sentences "Mr" "Smith and Mr" and "Jacob are best friends".

3. Punkt Sentence Tokenizer

The Punkt Sentence tokenizer is a bottom-up approach towards sentence segmentation. Punkt uses an Unsupervised algorithm to build a model for abbreviation words, collations and words that start sentences. Punkt can be pre-trained on an extensive collection of plain text available in the target language. For English NLTK data package already includes a pre-trained Punkt sentence tokenizer. Although the pre-trained model might be quite unsuitable when we are dealing with data from a specific domain. In such cases, we can use Punkt to learn parameters dynamically.

4. Naive vs Punkt

a. From the solution provided for the second question, it is evident that the naive algorithm for sentence segmentation using sentence delimiters can not handle abbreviations. In such a case the Punkt algorithm performs better as it has been pre-trained on a large dataset and hence can recognize abbreviations as part of the sentence.

b. The naive approach performs better when the query has inappropriate use of spaces at the end of sentences. The pre-trained Punkt model has been built on proper English texts which grammatically have space after a period. Therefore any typo in the text can not be dealt with Punkt. Sentences like " Good morning everyone.Hope you are doing well." Here Punkt will not be able to split the text into two sentences. Punkt might treat the period here as abbreviations due to its bottom-up knowledge. This can lead to arbitrarily long sentences if the user is not careful enough. However, The naive approach will split the text into two sentences as "Good morning everyone" and "Hope you are doing well".
Also, the pre-trained Punkt model can not be suitable for all domains. In the case of specific domains, Punkt can be trained dynamically but requires a significant amount of data which is often not present.

5. Word Tokenization

The simplest top-down approach to word tokenization is by using **spaces and symbols** like hyphens(-), commas(,), star marks (*), etc. to locate word endings. We can use the re.split function from the regular expression (re) python library to do this.

6. The Penn Treebank Word Tokenizer

The Penn Treebank tokenizer is a **top-down approach** for word tokenization that uses regular expressions as defined in Penn Treebank to tokenize text. It uses a direct set of rules for what constitutes a word **rather than learning from data**, and thus it's top-down nature.

It assumes that the input has been segmented into sentences. The Penn Treebank tokenizer performs the following steps:

1) Split standard contractions, e.g. ”isn’t” to ”is” and ”n’t”
2) Treat most punctuation characters as separate tokens.
3) Split commas and single quotes when followed by a whitespace.
4) Separate periods that appear at the end of line.

7. Comparison (Naive Top-Down vs Penn Treebank)

The Penn Treebank Tokenizer

It is better at handling standard contractions (compressed versions of words). For example, let the sentence to be tokenized be “I won’t try this”.

Penn Treebank:[”I”, ”wo”, ”n’t”, ”try”, ”this”]
Naive:[”I”, ”won”, ”t”, ”try”, ”this”]

Naive approach changes can’t to ’can’ and ’t’. Thus, the information about negation (i.e., cannot) is lost. On top of that the word “won” would also mean a completely different thing and the context of the sentence is also lost.
The Penn Treebank retains negation,as ”n’t” can be treated as ”not” for many practical purposes.

The Naive Approach

One part where the naive approach is superior to Penn Treebank is at managing hyphens(-). The Penn Treebank ignores hyphens altogether.
For example, let the sentence to be tokenized be : “The Woodland period of North American pre-Columbian cultures lasted from roughly 1000 BCE to 1000 CE..”

Penn Treebank : [”The”, ”Woodland”, ”period”, ”of”, ”North”, ”American”,”pre-Columbian”, ”cultures”, "lasted”, ”from”, ”roughly”, ”1000”, ”BCE”, ”to”, “1000” “CE.”]

Naive : [”The”, ”Woodland”, ”period”, ”of”, ”North”, ”American”, ”pre” ,”Columbian”, ”cultures”, "lasted”, ”from”, ”roughly”, ”1000”, ”BCE”, ”to”, “1000” “CE.”]

Words like ”pre-Columbian” may not be there in the dictionary which might present further issues.

8. ## Difference

For grammatical reasons, every document uses different forms of a word, such as submit, submitted, submission, submitting. There are families of derivationally related words having similar meanings, for example, democracy, democratic, and democratization. Both stemming and lemmatization are used to reduce inflectional forms and derivationally related forms of a word to obtain a common base form. Stemming is a crude heuristic process in which a word is chopped off from the ends assuming that the base form can be obtained by removing affixes. Lemmatization on the other hand makes use of a vocabulary and morphological analysis of words. It removes inflectional endings and returns the base or dictionary form of a word which is known as the lemma. Stemmers use language-specific rules, but they require less knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to correctly lemmatize words. Also, Lemmatization has higher latency when compared to stemming, as the query size increases it becomes a bottleneck. On the other hand, stemming can not reduce words like caught and bought to their base form catch and buy, which can be easily reduced by lemmatizer.

9. **Stemming or Lemmatization?**

The choice of stemming or lemmatization depends on the problem that we want to address and varies from case to case. There is an inherent precision-recall tradeoff. Stemming would attempt to increase the recall while lemmatization attempts to increase the precision of search.
Precision = (Retrieved and Relevant)/Retrieved
Recall = (Retrieved and Relevant)/Relevant
For a search engine, we would preferably need a higher precision. Out of all the retrieved documents, the maximum should be relevant. All the choices that the user gets should have relevant information and hence we decided to use lemmatization. Since we are having plain English text we can use lemmatization.

10.

```
class StopwordRemoval():

    def fromList(self, text):
        """
        Sentence Segmentation using the Punkt Tokenizer

        Parameters
        ----------
        arg1 : list
            A list of lists where each sub-list is a sequence of tokens
            representing a sentence

        Returns
        -------
        list
            A list of lists where each sub-list is a sequence of tokens
            representing a sentence with stopwords removed
        """

        for i in range(len(text)):
            final_tokens = []
            tokens = text[i]
            for W in tokens:
                if W not in stop_words:
                    final_tokens.append(W)
            text[i] = final_tokens
            # Replace the corresponding tokenized sentence with final_tokens

        return text
```

Done in the coding part of the assignment
Tokenized query: ["what", "similarity", "laws", "must", "be", "obeyed", "when", "constructing", "aeroelastic", "models", "of", "heated", "high", "speed", "aircraft"]
Reduced query: ["what", "similarity", "law", "must", "be", "obey", "when", "construct", "aeroelastic", "model", "of", "heated", "high", "speed", "aircraft"]
Can be seen here that words like laws, obeyed, constructing reduced to law, obey, construct.

11.

Done in the coding part of the assignment

```
class StopwordRemoval():

    def fromList(self, text):
        """
        Sentence Segmentation using the Punkt Tokenizer

        Parameters
        ----------
        arg1 : list
            A list of lists where each sub-list is a sequence of tokens
            representing a sentence

        Returns
        -------
        list
            A list of lists where each sub-list is a sequence of tokens
            representing a sentence with stopwords removed
        """

        for i in range(len(text)):
            final_tokens = []
            tokens = text[i]
            for W in tokens:
                if W not in stop_words:
                    final_tokens.append(W)
            text[i] = final_tokens
            # Replace the corresponding tokenized sentence with final_tokens

        return text
```

12. **Frequency Approach**

Data from different domains can be processed to detect the most frequently occurring words. The most obvious bottom-up approach for detecting a stop word list is to arrange all the words in the corpus by collection frequency that is the total number of times each word occurs in the document collection, and then to choose the most frequent words as stop words. Even this bottom-up approach will be augmented by some top-down approach while giving the frequency threshold to decide the most frequent words or as the number of clusters while clustering the frequently occurring words together.

References:

- https://docs.python.org/3/library/re.html
- https://www.nltk.org/api/nltk.tokenize.html
- https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-Stop-words-1.html