CS6370 Natural Language Processing

Jan – May 2021

Assignment 1 – Part 1

ME17B158 - Omkar Nath
ME17B170 – Uma T V

1. What is the simplest and obvious top-down approach to sentence segmentation for English Texts?

In English language, an **end mark** is a punctuation mark used at the end of a sentence to show that the sentence is finished. There are three end marks: the **period**, the **question mark**, and the **exclamation mark**. The simplest and obvious top-down approach to sentence segmentation for English Texts would be to separate sentences **based on end-marks**. Hence, a sentence would be the part of text in between the surrounding immediate end marks.

Reference:
https://www.englishgrammar101.com/module-12/punctuation-end-marks-and-commas/lesson-1/sentence-end-marks

2. Does the top-down approach (your answer to the above question) always do correct sentence segmentation? If Yes, justify. If No, substantiate it with a counter example.

No, the end mark separation technique **does not** always do correct sentence segmentation. As a counter example, the **dot symbol** used for **abbreviations** will be considered as an end mark and will strip out parts of sentences, making leaving the formed parts meaningless.

For example, the sentence “I am currently pursuing my B.Tech in Mechanical Engineering.” is separated as:

Sentence 1: I am currently pursuing my B. (meaningless)
Sentence 2: Tech in Mechanical Engineering. (meaningless)

3. Python NLTK is one of the most commonly used packages for Natural Language Processing. What does the Punkt Sentence Tokenizer in NLTK do differently from the simple top-down approach? You can read about the tokenizer here.

Unlike the end-mark separation technique which uses a top-down approach, the Punkt Sentence Tokenizer in NLTK uses a **bottom-up approach**. The Punkt Sentence Tokenizer uses an **unsupervised algorithm** to **build a model** for abbreviation words, collocations, and words that start sentences; and then uses that model to find sentence boundaries. It can be either **pre-trained** or we can train it on our available dataset **dynamically** and use the model for sentence segmentation.

Reference:
https://necromuralist.github.io/text-processing/sphinx/nltk/autogenerated/nltk.tokenize.punkt.PunktSentenceTokenizer.html

4. Perform sentence segmentation on the documents in the Cranfield dataset using:

Note: The codes for this question are attached and submitted along with the report.
Running the codes in command prompt

(a) The top-down method stated above

- Here, we just split sentences based on the three end marks (“.”,”?”,”!”)
- First, we check if there is an input,
- Then re package is used to split on these end marks
- White spaces at the edges of the sentences are then stripped.
- The result is then returned.

```
def naive(self, text):
    if isinstance(text, str):
        segments = re.split(delimiters, text)
        sentences = [s.strip() for s in segments]
        while '' in sentences:
            sentences.remove('')
        return sentences
    else:
        print("No text received")
        return ([])
```

Screenshot of the code written

```
D:\My Data\8th Sem\CS6370 Natural Language Processing\Assignment 1\ME17B158_ME17B170\Assigment_1_Code>python main.py
-segmenter naive -tokenizer naive -out_folder naive_segmentation_naive_tokenization\
```

The command prompt code for running a naive sentence segmentation code

(b) The pre-trained Punkt Tokenizer for English

- We use the pre-trained Punkt Tokenizer using the nltk.tokenize library
- First, we check if there is an input,
- Then a tokenizer of the package Punkt is defined.
- Sentences are tokenized using this tokenizer.
- The result is then returned.

```
def punkt(self, text):
    if (isinstance(text, str)):
        tokenizer = PunktSentenceTokenizer(text)
        sentences = tokenizer.tokenize(text)
        return sentences
    else:
        print("No text received")
        return ([])
```

A screenshot of the code

```
D:\My Data\8th Sem\CS6370 Natural Language Processing\Assignment 1\ME17B158_ME17B170\Assigment_1_Code>python main.py
-segmenter punkt -tokenizer naive -out_folder punkt_segmentation_naive_tokenization\
```

The command prompt code for running a punkt sentence segmentation code

State a possible scenario along with an example where:

(a) the first method performs better than the second one (if any)

Possible scenario when the first method performs better:

When we have **simple English sentences** ending with end marks and not involving dots for abbreviations, we can easily use the naive approach to get a computationally **fast** algorithm. However, using the second method would involve the pain of obtaining the corpus data and would be computationally very expensive.

Example document:

“My name is Riya. I am three years old.”

The first method gives: (Very fast results)
Statement 1: My name is Riya.
Statement 2: I am three years old.

The second method gives: (Computationally very expensive and requires obtaining corpus data)
Statement 1: My name is Riya.
Statement 2: I am three years old.

(b) the second method performs better than the first one (if any)

Possible scenario when the second method performs better:

When the presence of a dot used in **abbreviations** is taken to be a full stop and used to break into sentences.

Example document:

“I am currently pursuing my B.Tech in Mechanical Engineering.”

The first method gives:
Statement 1: I am currently pursuing my B. (Meaningless)
Statement 2: Tech in Mechanical Engineering. (Meaningless)

The second method gives:
Statement 1: I am currently pursuing my B.Tech in Mechanical Engineering. (Meaningful)

Clearly, the second method performs better than the first.

5. What is the simplest top-down approach to word tokenization for English texts?

Usually, the English language script either has spaces or word punctuation marks that denote the start and end of words. Examples of word punctuation marks are comma (,) , hyphen (-) , front slash (/) etc. Hence, the simplest top-down approach to word tokenization would be to separate words **based on spaces and word punctuations**. Hence, a word would be the part of text in between the surrounding immediate spaces or word punctuations.

References:
https://www.theidioms.com/punctuation/

6. Study about NLTK's Penn Treebank tokenizer here. What type of knowledge does it use – Top-down or Bottom-up?

NLTK's Penn Treebank Tokenizer uses **Top-down knowledge** to segment words. It has a **list** of expressions and the tokenization solutions that it uses to Tokenize words which overcome some of the limitations of the simple approach of segmentation based on spaces and word punctuations.

This tokenizer performs the following steps:

- split standard contractions, e.g. don't -> "do" and "n't" and they'll -> "they" and "'ll"
- treat most punctuation characters as separate tokens
- split off commas and single quotes, when followed by whitespace
- separate periods that appear at the end of line

References:
https://www.kite.com/python/docs/nltk.TreebankWordTokenizer

7. Perform word tokenization of the sentence-segmented documents using

Note: The codes for this question are attached and submitted along with the report.
Running the codes in command prompt

(a) The simple method stated above

- Here, we perform tokenization based on punctuations.
- Firstly, we check for an input.
- Then the given string is split on the chosen punctuation marks.
- Edge cases are removed (black words, symbols) and the result is returned.

```
def naive(self, text):
    tokenized = []
    if isinstance(text, list):
        for s in text:
            if isinstance(s, str):
                token = re.split(word_separators, s)
                for word in token:
                    if ((word in punctuations) or (word ==' ') or (word =='')):
                        token.remove(word)
                tokenized.append(token)
    else:
        print("No text received")
    return tokenized
```

Screenshot of the code.

```
D:\My Data\8th Sem\CS6370 Natural Language Processing\Assignment 1\ME17B158_ME17B170\Assigment_1_Code>python main.py
-segmenter naive -tokenizer naive -out_folder naive_segmentation_naive_tokenization\
```

The command prompt code for running a naive tokenizer code

```
punctuations = ['\'', '\"', '?', ':', '!', '.', ',', ';', '&', '#', '(', ')', '[', ']', '{', '}', '_', '|']
```

The chosen punctuation seprators.

(b) Penn Treebank Tokenizer

- Here, we perform tokenization using the Penn Tree Bank tokenizer of the nltk.tokenize library
- Firstly, we check for an input.
- Then for every sentence, a tokenizer of nltk package is used to tokenize the sentence.
- Results are returned

```
def pennTreeBank(self, text):
    tokenized = []
    if isinstance(text, list):
        for s in text:
            if isinstance(s, str):
                token = TreebankWordTokenizer().tokenize(s)
                tokenized.append(token)
    else:
        print("No text received")
    return tokenized
```

Screenshot of the Code

```
D:\My Data\8th Sem\CS6370 Natural Language Processing\Assignment 1\ME17B158_ME17B170\Assigment_1_Code>python main.py
-segmenter naive -tokenizer ptb -out_folder naive_segmentation_ptb_tokenization\
```

The command prompt code for running a ptb tokenizer segmentation code

State a possible scenario along with an example where:

(a) the first method performs better than the second one (if any)

The first method does not involve use of corpus data and is computationally cheaper than the second. Hence, it performs better for **simple text data**.

Example document: I like apples,oranges,bananas and grapes.

First Method: [“I”,”like”,”apples”,”oranges”,”bananas”,”and”,”grapes”]
(**Fast** and does not require corpus data)

Second Method: [“I”,”like”,”apples”,”oranges”,”bananas”,”and”,”grapes”]
(**Computationally expensive** and requires corpus data)

(b) the second method performs better than the first one (if any)

The second method is also a top-down like the first one, it however is a better version which **handles the limitations** of the first method. The first method blindly tokenizes words based on punctuations; it does not take into account the **limitations of the English language.**

Example: I don’t smoke.

First Method: [“I”,”don”,”t”,”smoke”] (“don” and “t” are meaningless)
Second Method: [“I”,”do”,”n’t”,”smoke”] (all words make sense)

However, there are still limitations of this. It is possible that we may have some informative symbols like $ which are meant to convey some information. The Penn Treebank tokenizer may remove such symbols, which may not be to our interest. The naïve approach does not suffer from this limitation.

8. What is the difference between stemming and lemmatization?

Stemming and Lemmatization are text normalization techniques within the field of Natural language Processing that are used to prepare text, words, and documents for further processing.

Stemming is the process of producing **morphological variants of a root/base word**. It is a crude method for cataloging related words which **chops off letters from the end until the stem is reached**. Using this, while searching text of a certain keyword, even the search results of the variants of the word can be obtained. For example, searching for “boat” may also return “boats” and “boating”. However, English has many exceptions where a more sophisticated process is required.

In contrast to stemming, lemmatization **looks beyond word reduction and considers a language's full vocabulary** to apply a morphological analysis to words. The lemma of 'was' is 'be' and the lemma of 'mice' is 'mouse'. Lemmatization is much more informative than simple stemming. Lemmatization **looks at surrounding text** to determine a given word's **part of speech**.

References:
https://towardsdatascience.com/stemming-vs-lemmatization-2daddabcb221

9. For the search engine application, which is better? Give a proper justification to your answer. This is a good reference on stemming and lemmatization.

For a search engine application, **Lemmatization** is better.

A search engine must be able to handle all those particular **morphological complexities** while searching for words. It is common for the forms of words to change based on how they are used. Typically, most search engines and search solutions normalize by "stemming" or "lemmatization".

Stemming is a crude method of chopping off characters at the end of a word in the attempt to find the root word. There are a lot of limitations of this method.

For example, the user searches with the word "celebrities", as in the plural of celebrity, but the search engine ends up with a stem of "celebr". That search could end up with false positives from other words with the same stem as "celebrations". The only way to truly get **accurate** results is to perform a more **advanced morphological analysis** to find the "lemma" or dictionary form of the word— this is called "lemmatization".

Taking the previous example, "celebrities" is searched, but with lemmatization utilized by the search engine, the query is correctly interpreted as "celebrity", not "celebration", enabling the search engine to deliver the right results.

Stemming gives better recall and **Lemmatization gives better precision**. Since the number of correct results is very large, having **better precision** is better for search engines. Hence, Lemmatization works better on search engines.

10. Perform stemming/lemmatization (as per your answer to the previous question) on the word-tokenized text.

- Here, we perform Lemmatization on the tokenized words using the nltk library
- First, we check if there is an input.
- For each sentence, blank word are removed.
- Then for each word in each sentence, the part of speech is determined using the library and a custom defined function.
- The word is then appropriately lemmatized.
- The result is returned.

```
class InflectionReduction:
    def reduce(self, text):
        if isinstance(text, list):
            for s in range(len(text)):

                while '' in text[s]:
                    text[s].remove('')
                pos_tags = pos_tag(text[s])

                for word in range(len(text[s])):
                    pos = convert_to_wordnet(pos_tags[word][1])
                    text[s][word] = wordnet_lemmatizer.lemmatize(text[s][word], pos=pos)
            return text
        else:
            print("No text received")
            return []
```

Screenshot of the function code.

```
def convert_to_wordnet(s):
    if s.startswith("J"):
        return wordnet.ADJ
    elif s.startswith("V"):
        return wordnet.VERB
    elif (s.startswith("N")) | (s.startswith("P")):
        return wordnet.NOUN
    elif s.startswith("R"):
        return wordnet.ADV
    else:
        return wordnet.NOUN
```

Function in util.py to determine the Part of Speech

11. Remove stopwords from the tokenized documents using a curated list of stopwords (for example, the NLTK stopwords list).

- We remove the stopwords from the lemmatized set of words using the nltk library
- Only those words not in stop words are considered.
- Stop words are taken from the ntlk library of python.

```
class StopwordRemoval():

    def fromList(self, text):
        for s in range(len(text)):
                token_words = []
                tokens = text[s]
                for word in tokens:
                        if word not in stop_words:
                                token_words.append(word)
                text[s] = token_words
        return text
```

Screenshot of the code

12. In the above question, the list of stopwords denotes top-down knowledge. Can you think of a bottom-up approach for stopword removal?

Stopwords are words in sentences that contain very **little information** and have the **least discriminating ability**. Some examples of stopwords are articles, prepositions, determinants etc. Another corollary property of stopwords is that they also occur **more frequently** as compared to the discriminating words. Hence, a bottom-up approach for stopword removal would be to calculate the frequencies of every word in the corpus and choose the words with **frequencies above a threshold** as stopwords.

This is a common standard approach used in many bottom-up stopword removal methods: We first compute the **Inverse Document Frequency.**

$$IDF = \log (n/N)$$

Where N = total number of docs in collection

n = number of documents in which the term occurs

The **weight** for every word is computed as:

$$W_{t,di} = \text{term frequency (local measure)} * \text{IDF (global measure)}$$

The words having **weights above a threshold** are considered stopwords.

Reference:
Class notes
https://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html