CS6370 Assignment1 part1

Saili Myana (EE18B062), Sai Rithvik (EE18B051)

11th March, 2021

Introduction

We aim to build a search engine from scratch, which is an example of an information retrieval system. This is the first part of Assignment 1, where we will be building a basic text processing module that implements sentence segmentation, tokenization, stemming/lemmatization, and stopword removal using the Cranfield Dataset.

1. Simplest top-down approach for Sentence Segmentation

In general, a sentence may end with a period, question mark, or exclamation. So, the most straightforward top-down approach for sentence segmentation is to find where the sentence ends by looking for a period (.), question mark (?), or exclamation (!) and segmenting the sentence using these as delimiters.

2. Will this top-down approach always work?

No, the above top-down approach will not always segment the sentence correctly. When there are standard abbreviations that end in a period (.), say Mr. or Inc., our model considers it as the end of a sentence.
An example for which the above approach fails is:
{*'My Home Constructions Inc. has a high turnover. What about your startup?'*}
The output is going to be:
{*'My Home Constructions Inc. ', 'has a high turnover. ', 'What about your startup?'*}

3. What does the Punkt Sentence Tokenizer in NLTK do differently?

Punkt tokenizer divides a text into a list of sentences using an unsupervised algorithm to build a model for abbreviation words, collocations, and words that start sentences. It must be trained on a large collection of plaintext in the target language before it can be used. Whereas in the simple top-down approach, we blindly look for delimiters(. ! ?) to segment the sentences. Though this approach works for simple sentences, these delimiters need not always be the end-of-sentence. Punkt algorithm rectifies this by learning the start-of-sentences from a large dataset.
A Punkt trainer can be used for incremental training and modification of hyperparameters to identify the start of sentences. We are using a pre-trained Punkt tokenizer from NLTK.

4. Performing sentence segmentation

a) The top-down method

```
def naive(self, text):
    """
    Sentence Segmentation using a Naive Approach
    Parameters
    ----------
    arg1 : str
      A string (a bunch of sentences)
    Returns
    -------
    list
      A list of strings where each string is a single sentence """
    segmentedText = None
    #Fill in code here
    segmentedText = re.findall('.*?[!?.]\s*', text)
    return segmentedText
```

b) The pre-trained Punkt Tokenizer for English

```
def punkt(self, text):
    """
    Sentence Segmentation using the Punkt Tokenizer
    Parameters
    ----------
    arg1 : str
      A string (a bunch of sentences)
    Returns
    -------
    list
      A list of strings where each strin is a single sentence
    """
```

```
    segmentedText = None
    #Fill in code here
    sent_detector=nltk.data.load('tokenizers/punkt/english.pickle')
    segmentedText=sent_detector.tokenize(text.strip())
    return segmentedText
```

Comparing the two methods of Sentence Segmentation

a) The scenario where simple top-down approach performs better than PUNKT tokenizer: The sentence tokenizer of Punkt accounts for most cases but fails when the query has inappropriate grammar or punctuation.
For the below sentence:
{*'He is your friend.You should try to help him! Could you talk to the teacher?'*}
The naive model returns:
{*'He is your friend.', 'You should try to help him! ', 'Could you talk to the teacher?'*}
Whereas the Punkt sentence tokenizer returns:
{*'He is your friend.You should try to help him!', 'Could you talk to the teacher?'*}
As we can see, the Punkt tokenizer combines the first two sentences into a single token, as it splits the sentence only when there's a space after a period. But our tokenizer splits at all periods, so it works in this case.

b) The scenario where PUNKT Tokenizer approach perform better than simple top-down approach:
The top-down approach, as expected, fails on the trivial cases of abbreviations that end in a period. For the example discussed in question 2:
The simple top-down model returns the following:
{*Home Constructions Inc. ', 'has a high turnover. ', 'Did u know that? '*}
But the Punkt algorithm correctly returns:
{*'My Home Constructions Inc. has a high turnover.', 'Did u know that? '*}
This is expected, as our model always segements at periods, and other end-of-sentence punctuation, but the Punkt tokenizer makes a decision at each punctuation.

5. Simplest top-down approach for word tokenization

The simplest approach for word tokenization is to split the sentence with space, comma (‘,’), colon (‘:’), semi-colon (‘;’), hyphen (‘-’), and the end of sentence markers we used for Naive sentence segmentation (. ? !) as delimiters. We segment the sentence at each instance of these punctuation marks.

6. Penn Treebank Tokenizer

NLTK's Penn Treebank (PTB) tokenizer uses a top-down approach. This can be inferred from the fact that the PTB tokenizer uses a list of rules (background knowledge) to tokenize each word from a sentence. Some of these rules include: split standard contractions like $don't \longrightarrow do\ n't$ and $They'll \longrightarrow They\ 'll$.

7. Performing word tokenization of the Sentence-Segmented documents

a) The simple method

```
def naive(self, text):
    """
    Tokenization using a Naive Approach

    Parameters
    ----------
    arg1 : list
      A list of strings where each string is a single sentence

    Returns
    -------
    list
      A list of lists where each sub-list is a sequence of tokens
    """
    tokenizedText = None
    #Fill in code here
    tokenizedText = []

    for i in text:
      tokenizedText.append([temp for temp in re.split(r'([\s
    ,:;?!.-])\s*', i) if temp!=' ' and temp])
    return tokenizedText
```

b) Penn Treebank Tokenizer

```
def pennTreeBank(self, text):
    """
    Tokenization using the Penn Tree Bank Tokenizer

    Parameters
    ----------
    arg1 : list
      A list of strings where each string is a single sentence

    Returns
    -------
    list
      A list of lists where each sub-list is a sequence of tokens
    """
```

```
15
16      tokenizedText = None
17
18      #Fill in code here
19      tokenizedText = []
20
21      for i in text:
22        tokenizedText.append(TreebankWordTokenizer().tokenize(i))
23      return tokenizedText
```

Comparing two methods of word segmentation

a) Scenario where simple method performs better than Penn Treebank Tokenizer: Penn Treebank tokenizer ignores hyphens (-), whereas our simple model splits the words at hyphens. This is useful, as we can compare the individual words of the compound noun with the documents. Works even if the entire compound noun is not present in the document.
For the text:
{'The company reported a two-fold rise after their high-tech solution.'}
Our simple tokenizer outputs:
{'The', 'company', 'reported', 'a', 'two', '-', 'fold', 'rise', 'after', 'their', 'high', '-', 'tech', 'solution', '.'}
Whereas the PTB tokenizer outputs the following:
{'The', 'company', 'reported', 'a', 'two-fold', 'rise', 'after', 'their', 'high-tech', 'solution', '.'}

b) Scenario where Penn Treebank Tokenizer performs better than our simple method:
Our simple tokenizer fails for contractions like "can't" and "don't", and considers the whole word as a single token.
For the text:
{'We won't be able to participate in the competition, we didn't like the rules.'}
Our simple tokenizer outputs:
{'We', 'won't', 'be', 'able', 'to', 'participate', 'in', 'the', 'competition', ',', 'we', 'didn't', 'like', 'the', 'rules', '.'}
Whereas the PTB tokenizer outputs the following:
{'We', 'wo', 'n't', 'be', 'able', 'to', 'participate', 'in', 'the, 'competition', ',', 'we', 'did', 'n't', 'like', 'the', 'rules', '.'}
This type of tokenization is helpful, as it retains the information about both the words (negation and root word).

8. Stemming and Lemmatization

- Stemming is a method in which inflected words and derivational affixes get converted into their root forms in a crude manner. Different inflections and derivations return the same word, ignoring the Parts of Speech(POS) of the words.

- Whereas in lemmatization, inflected words are converted into their root words with the morphological analysis of words. POS is preserved in lemmatization.

For example:
The words "dance", "dancing" and "danced" get converted to "danc" by stemming but gets converted to "dance" using lemmatization.

9. For Search Engine Application, which is better?

- Stemming allows us to return a wide range of documents for a query. This means, when we chose stemming for our application, we give priority to recall over precision.

- Though both stemming and lemmatization perform well on Information Retrieval tasks, we chose stemming for our toy search engine. This will make our search engine retrieve additional documents, and we can rank them appropriately in the later stages.

- The precision-recall tradeoff usually decides the inflection-reduction module of the NLP application. Lemmatization could be used in particular search engines when precision has to be optimized.

10. Stemming on word-tokenized text

```
def reduce(self, text):

    """
    Stemming/Lemmatization

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

    Returns
    -------
    list
      A list of lists where each sub-list is a sequence of
      stemmed/lemmatized tokens representing a sentence
    """

    reducedText = None

    #Fill in code here

    reducedText = []
    ps = PorterStemmer()

```

```
    for i in text:
      list1 = []
      for token in i:
        list1.append(ps.stem(token))
      reducedText.append(list1)

    return reducedText
```

11. Removal of Stop Words

```
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
    """

    stopwordRemovedText = None

    #Fill in code here
    stopwordRemovedText = []
    stop_words = set(stopwords.words('english'))

    for sent in text:
      list1 = []
      for token in sent:
        if token not in stop_words:
          list1.append(token)

      stopwordRemovedText.append(list1)

    return stopwordRemovedText
```

12. Bottom-up approach for Stop word removal

We can identify the most frequently occurring words from the corpus and store them as the stop words. We could then traverse over the text to remove these. This method assumes no background knowledge and extracts stop words from the given text data. This would be a simple bottom-up approach to find and remove the stop words from our text.

Conclusion

In this assignment, we have shown how to perform sentence segmentation by a naive, simple, top-down approach and using pre-trained PUNKT tokenizer, word tokenization using a simple top-down Penn Treebank Tokenizer, stemming, and finally stopword removal from raw text data. We have also argued why we chose stemming for our task of Information retrieval.

References

1. Sentence Segmentation, Pg.30, Speech and Language Processing
(D. Jurafsky, James.H.Martin)
2. PUNKT Sentence Tokenizer in NLTK library
3. Penn Treebank Tokenizer in NLTK library
4. re: Regular expression operations, Python 3.9.2 documentation
5. Stemming and Lemmatization - Stanford