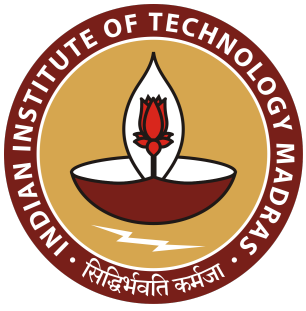
Indian Institute of Technology, Madras

CS6370: Natural Language Processing - Report
Assignment 1, Part 1

Nithin Uppalapati

EE18B035
ee18b035@smail.iitm.ac.in

D Krithin Chowdary
EE18B047
ee18b047@smail.iitm.ac.in



15-03-2021

Contents

1. Aim

The goal of the assignment is to build a search engine from scratch, which is an example of an information retrieval system. In the class, we have seen the various modules that serve as the building blocks of a search engine. We will be progressively building the same as the course progresses. The first part of this assignment is to build a basic text processing module that implements sentence segmentation, tokenization, stemming/lemmatization and stopword removal. The Cranfield dataset will be used for this purpose, which has been uploaded separately on Moodle.

2. Answering the Questions Given in Assignment (A1-Part1)

2.1 What is the simplest and obvious top-down approach to sentence segmentation for English texts?

The simplest and obvious top-down approach for segmentation of English text is to break the sentences at suitable punctuation marks. For this we define a set of punctuation marks which imply *(not true in practise)* the termination of a sentence i.e {. , ! , ?}. We search for these punctuation marks in the text and then segment the sentences accordingly.
One point which is worth mentioning is: not to segment sentence at new line characters. As newline character need not imply the sentence termination.

2.2 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 top-down approach mentioned above fails to do proper sentence segmentation in some cases. One such instance is when "**.**"(period) is not used to terminate a sentence.(Cases like: when we use periods in abbreviations; when we use honorifics like - Mr. , Mrs. , etc.)
Ex:- Mrs. Smith, F.B.I, I.I.T etc.

2.3 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?

- Punkt tokenizer is not a top-down implementation, it is bottom-up approach. Punkt has the trained data which helps it to differentiate abbreviations and honorifics from the "actual" sentence termination.

- Punkt is designed to learn parameters (a list of abbreviations, etc.) unsupervised from a corpus similar to the target domain.

2.4 Perform sentence segmentation on the documents in the Cranfield data set using:

2.4.1 The top-down method stated above

In this method we use a for loop to traverse through the text to find out specific punctuation marks and then we segment them at those locations using the inbuilt function split(). It's done using the following piece of code.

```
[cnt,k]=[0,0]
lst=[]
s={".","?","!"}
for i in text:
  if i in s:
    senten=text[k:cnt+1].strip()
    lst.append(senten)
    k=cnt+1
  cnt=cnt+1
```

2.4.2 The pre-trained Punkt Tokenizer for English

For this method we load the Punkt tokenizer which has been pre-trained on a dataset and then use this to segment the required text. This is done using the following code.

```
sent_detector = nltk.data.load('tokenizers/punkt/english.pickle')
segmentedText=sent_detector.tokenize(text.strip())
```

2.4.3 State a possible scenario along with an example where:

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

1. This implementation performs correctly even if there is no space given after the punctuation symbols like - "**.**" , "**!**", "**?**"
    - In Punkt Method the segmentation does not occur if there was no space followed by the punctuation mark. (results shown below)

```
>>> import nltk.data
>>> from nltk.corpus import stopwords
>>> from nltk.tokenize import word_tokenize
>>> from nltk.stem import PorterStemmer, WordNetLemmatizer
>>> sent_detector = nltk.data.load('tokenizers/punkt/english.pickle')
>>> text="Hi, this is Nithin and Krithin, teammates for NLP assignments.We
... both are undergrads in EE, at IIT Madras!Do you ever wondered why our
```

```
... names are similar?We too don't have an answer for that!"
>>> segmentedText=sent_detector.tokenize(text.strip())
>>> segmentedText
    ["Hi, this is Nithin and Krithin, teammates for NLP assignments.We
... both are undergraduates in EE, at IIT Madras!Do you ever
... wondered why our names are similar?We too don't have an
... answer for that!"]
>>> len(segmentedText)
    1 #No segments are made, when the above text is passed (Punkt method)
```

- Whereas, in our implementation (for the above text) the segmentation occurs as desired. (results shown below)

```
>>> segmentedText
    ['Hi, this is Nithin and Krithin, teammates for NLP assignments.', 'We both
... are undergraduates in EE, at IIT Madras!', 'Do you ever wondered
... why our names are similar?', "We too don't have an
... answer for that!"]
>>> len(segmentedText)
    4 #4 segments are made, when the above text is passed (naive method)
```

2. Time taken to execute

- Below is a demonstration of comparison of times taken by naive approach and Punkt approach. The naive approach is faster than Punkt approach as the naive approach is just an algorithm, whereas the punkt approach involves loading (reaading) trained datasets like - *english.pickle* file.

- Below is code for measuring the time taken by all four possible combinations ({naive,naive}, {Punkt,naive}, {naive,PennTB}, {Punkt,PennTB}) while operating on cranfield dataset

```
import time
begin = time.time()
### execute main.py repeatedly for "x" times
end = time.time()
print((end-begin)/x)

## Results are below: for x = 1000
>>> -segmenter naive -tokenizer naive
... 0.13249667239189147
>>> -segmenter punkt -tokenizer naive
```

```
... 0.13617236471176147
>>> -segmenter naive -tokenizer ptb
... 0.16570245599746705
>>> -segmenter punkt -tokenizer ptb
... 0.16851172709465026
```

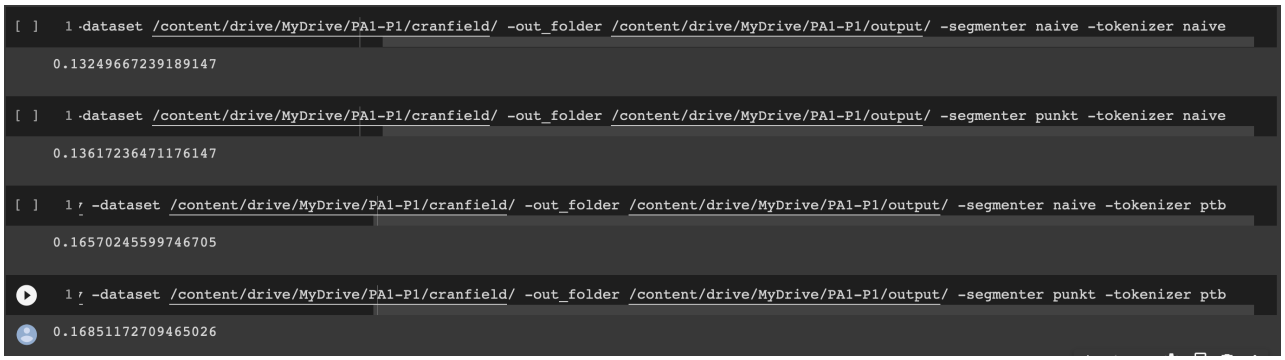


Figure 2.1: Iterating the program for 1000 times in Google Colab

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

1. Distinguishing abbreviations and honorifics from sentence termination
   - Punkt will not segment F.B.I. into "F.", "B.", "I." as it is not a sentence. Whereas the naive approach implemented above will segment these kind of cases, when it shouldn't actually do it.
   - Similarly the naive approach segments - "Mr. President" into "Mr.", "President" which is not desired. Whereas this issue will not occur with Punkt segmenter.
2. Punkt tokenizer is not just an algorithm, it can be trained from dataset. This is an unsupervised learning and hence, we need not worry about coding the corner cases in our algorithm. Whereas, the naive approach is just an algorithm.

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

The simplest top-down approach for word tokenization of English text is to split words separated by the white-space delimiter (" ") and the new-line ("\n")delimiter.

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

TreeBank is a syntactically annotated corpus where every sentence is paired with a corresponding tree. The Penn Tree Bank Project has treebanks from the Brown, Switchboard, Wall Street Journal corpora of English, and this dataset is maintained by the University of Pennsylvania. Type of knowledge used by PTB is Top-Down in nature. We invoke this tokenizer by **word_tokenize()**

2.7 Perform word tokenization of the sentence-segmented documents using:

2.7.1 The simple method stated above

We first use for loops to traverse through each string and split the words separated by space and new line delimiters. This is done with the help of split() function as shown in the code below.

```
lst=[]
for i in text:
    i.replace("\n"," ")
    lst.append(i.split(" "))
```

2.7.2 Penn Treebank Tokenizer

We import the word_tokenize function from the nltk.tokenize library and use this to tokenize the strings into words as done in the code below.

```
from nltk.tokenize import word_tokenize
l = []
    for i in text:
        w = word_tokenize(i)
        l.append(w)
```

2.7.3 State a possible scenario along with an example where:

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

1. Time taken to execute

   - Below is a demonstration of comparison of times taken by naive approach and PTB approach. The naive approach is faster than PTB approach as the naive approach is just an algorithm, whereas the PTB approach involves loading (reading) datasets. Refer Figure 2.1 to see timing analysis.

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

1. PTB tokenizer will tokenize short forms like: they'll to ["they"," 'll "] and so on...
   Whereas the naive method does not bother about symbols like **apostrophe**, hence the naive tokenizer fails at tokenizing theses short forms.

2.8 What is the difference between stemming and lemmatization?

- Stemming works by cutting off the starting or the ending of a given word based on a predefined set of prefixes and suffixes. Ex:- -cation, -able (justification and justifiable are cut down to justifi)
- Lemmatization considers the morphological analysis of words with the help of an algorithm which looks into dictionaries. Ex:- catching and caught are associated with the same lemma "catch".

Below is the table which summarizes the differences between the two methods:

Differences between Lemmatization and Stemming		
S.No	Lemmatization	Stemming
1	The root word is must be a morphological word (it belongs to language)	The root word need not necessarily be a morphological word (not necessarily belong to language)
2	Has more precision (than stemming)	Has more recall (Than lemmatization)
3	We can also specify parts of speech for acquiring better context of the root word	It just truncates the word accordingly to the mentioned rules in algo.
4	It depends on corpus or "detailed dictionary", WordNet...	It is just an algorithm which runs on predefined rules
5	It is slower when compared to stemming	It is faster when compared to lemmatization
6	Root Word is called "lemma"	Root word is called "stem"

2.9 For the search engine application, which is better? Give a proper justification to your answer.

For Search Engine application we prefer Lemmatization over Stemming.
We think lemmatization is a better option. In many cases stemming returns a lot of irrelevant documents (ex:- universe and university have the same root). Stemming also fails in the case of a few past tenses (Ex:- see and saw, catch and caught). Over stemming might occur in stemming which might reduce the relevance of the search results. These can be avoided by using lemmatization because it is aided by a source of vocabulary. In general, lemmatization

offers better precision than stemming, but at the expense of recall. It might be a bit time and resource consuming, but with recent advances in technology those extra computation timings are comparatively negligible.

2.10 Perform stemming (or) lemmatization (as per your answer to the previous question) on the word-tokenized text

Below is the implementation of lemmatization on the tokenized cranfield dataset, by using nltk module:

```
from nltk.stem import WordNetLemmatizer
wrntl=WordNetLemmatizer()
temp=text.copy()
for i in range(0,len(text)):
  for j in range(0,len(text[i])):
    temp[i][j]=wrntl.lemmatize(text[i][j])
reducedText = temp.copy()
```

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

For removing the stopwords we import the predifined set of stopwords from the nltk corpus and then use a for loop to eliminate any stopwords present in the tokenized dataset. The implementation of stopword removal on the tokenized cranfield dataset is given below.

```
from nltk.corpus import stopwords
for i in text:
  for j in stwords:
    while(i.count(j)):
      i.remove(j)
  stopwordRemovedText.append(i)
```

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

In bottom-up approach for removing the stop words, one outstanding feature of stop words is that they have high frequency of occurrence. So, we first analyse the frequency of the words. Another characteristic of a stop word been considered is that it is homogeneously distributed

across documents. The higher a word is spread across documents, the higher the chances of it being a stop word.
We can also perform entropy measure method, entropy helps in determining stop words based on the amount of information that a word carries. According to information theory, stop words are words that carry little information.

3. References

1. Reference on Stemming and Lemmatization (Click Here)
2. Reference on PTB
3. Tijani, Olatunde & Onashoga, Saidat & A.T., Akinwale. (2017). An Auto-Generated Approach Of Stop Words Using Aggregated Analysis.