CS6370 Natural Language Processing - Assignment - Part 1

Name: Atishay Ganesh , Dhruv Chopra
Roll Number: EE17B155 , EE16B107

February 28, 2020

1 Abstract

The goal of the first part of the assignment is to implement basic text processing functions, which include

- Sentence Segmentation
- Tokenization
- Inflection Reduction
- Stopword Removal

Top-down and bottom-up approaches are considered for the first two tasks.

2 Assignment

2.1 Implementation Details

We use nltk in Python 3.6 for implementing the various text processing tasks.

2.2 Answers to the questions

2.2.1

The simplest top-down approach to segment sentences is to split the document at the occurrence of sentence delimiters. Sentence Delimiters include . ? ! \n (newline) and sometimes ” or]. In general, the sentence delimiter will be followed by whitespace or an end of document character.

2.2.2

No, a simple top down approach for sentence segmentation will not always work for various reasons. The simple top down method will fail when there are sentence delimiting characters that appear in the middle of sentences (for example: e.g., i.e, in., etc.). It will produce a substantial number of false positives. Consider this sentence "He graduated with a Ph.D. from Caltech." The naive tokenizer would split it into at least 2 sentences (after Ph.D.) while it should be only 1.

2.2.3

While the simple top-down approach only looks at sentence delimiters, the Punkt Sentence Tokenizer uses an unsupervised algorithm to build a model for abbreviation words, collocations, and words that start sentences. It is a bottom-up approach that relies on data (along with a bit of top down knowledge) to determine sentence boundaries. Since it is trained in English language, it does not identify certain abbreviations in the text (for example in.), which is a problem with the naive method as well.[3]

2.2.4

(a) The Punkt Tokenizer has a tendency to create quite a few false negatives. (not splitting where it should split). For example consider the following string from a document in the corpus. "a free-flight investigation of ablation of a blunt body to a mach number of 13 .1. a five-stage rocket-propelled research-vehicle system was flown" The Punkt Tokenizer fails to realize that a new sentence starts at "a" (partially due to lack of capitalization). The naive method however works fine.

(b) As mentioned above, a sentence with delimiters used in the middle of a sentence will cause issues with the naive method. For example consider the following string which is from a document in the corpus. "It was o. reynolds who first expressed the so-called apparent or turbulent stresses by the mean values of the products of the velocity components." The naive method will create a new Sentence from "reynolds" but the Punkt Tokenizer will succeed.

Considering this, it is easy to cook up an example string where both methods fail, but at different places. Consider this string: "the blade designed by o. ram had length in cm of 1. his contributions to science were amazing." The naive method splits it into 3 sentences and the Punkt tokenizer says it is only 1 sentence. The ground truth, however, is that there are 2 strings in the sentence.

2.2.5

Similar to the naive method for sentence tokenization the simplest method for word tokenization is to split sentences according to certain delimiters, which include, ; , *
s (whitespace) : ' - "

2.2.6

The NLTK tokenizer splits sentences into word tokens based on a set of rules like separating contractions, treating punctuation as separate tokens, splitting off commas and single quotes when followed by a whitespace and separating periods at the end of a line. So, it only uses top down knowledge to tokenize a sentence.[4]

2.2.7

(a) The Penn Treebank Tokenizer does not split words with hyphen in between as two different words. However, the delimiter based naive approach splits them into two different words because hyphen has been explicitly added as a delimiter.

(b) The delimiter based approach does not split all contractions properly. For example, don't gets split as don and t but it should get split as do and nt which the Penn Treebank Tokenizer does.

2.2.8

Stemming is a heuristic process that follows a set of rules and known suffixes to reduce inflectional forms to stems or base forms. It is a fast and aggressive method, and often results in words not having their full On the other hand, a lemmatizer does a full morphological analysis on a word to reduce it to its base form. It uses a comprehensive dictionary to ensure that all stems are valid dictionary words.[5]

2.2.9

The stemmer is more suitable for a search engine application for a variety of reasons.

- Since the final decision is made by a human reader, the recall is extremely crucial. The required document must be chosen, but it is okay if a few other documents are there as well. Since the stemmer favours recall over precision, it is better suited for the task.

- Regarding a search engine application, we are interested in clustering words that are inflectionally related together, and hence reducing the

dimension of the problem. It does not really matter whether the words are grammatically correct.

- In the context of information retrieval, it has been observed that neither form of normalization improves English information retrieval performance in aggregate. i.e, both methods help some queries while doing badly on a lot of queries.[5]
- The lemmatizer is expected to be more accurate, while quite slower. However to be accurate it would require POS tagging, which would make it even slower.
- The stemmer is expected to be less precise and faster. The loss of precision among words is accepted as it performs well enough in practice.[2]
- In part due to the time required to load the dictionary to the memory, it takes about 63 times as long to lemmatize a query as it does to stem one (1157 ms vs 18 ms). But this is a one time cost (in case we are reducing multiple queries or documents at once), since you only load it once per execution of the code.

2.2.10

Code Attached

2.2.11

Code attached.

2.2.12

A naive bottom-up approach we could follow is removing words with high frequency and removing the words with a low inverse document frequency. Another approach that could be used is a supervised method, where we compute the mutual information between a word and a document class, which can provide knowledge about how much information a word contains about a given class. Consequently, words with low mutual information can be removed. In another method, we iterate over randomly selected separated chunks of data from web documents and rank terms in each chunk based on their in-format values using the Kullback-Leibler divergence between the following probability distributions: the normalized term frequency of a word within a chunk and the normalized term frequency of a word in the entire collection. Then we create the final stopword list by taking the least informative terms in all chunks by removing all possible duplications.[1]

References

[1] *A Systematic Review on Stopword Removal Algorithms*. Apr. 2018. URL: http://www.ijfrcsce.org/download/browse/Volume_4/April_18_Volume_4_Issue_4/1524218332_20-04-2018.pdf.

[2] Jirka. *Stemmers vs Lemmatizers*. June 2013. URL: https://stackoverflow.com/questions/17317418/stemmers-vs-lemmatizers.

[3] *Source code for nltk.tokenize.punkt*. URL: https://www.nltk.org/_modules/nltk/tokenize/punkt.html.

[4] *Source code for nltk.tokenize.treebank*. URL: https://www.nltk.org/_modules/nltk/tokenize/treebank.html.

[5] *Stemming and lemmatization*. Apr. 2009. URL: https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html.