CS6370 Assignment 1

Subhankar Chakraborty and S Sivasubramaniyan

February 2020

1 Sentence Segmentation

The most straightforward and apparent top-down approach to sentence segmentation for English texts would be the use of common sentence-delimiting punctuations like '.', '?' , '!', etc. to mark the end of a sentence. So whenever we encounter these characters while reading through a text, the preceding text encountered between the previous delimiter to the current one can be stored as a separate sentence.

Does the Top-Down approach always work?

The top-down approach has its obvious flaws. The clearest one being that it does not distinguish between the *'dot'(.)* used as a period and the one being used to denote an abbreviation. So the above top-down approach segments sentences like *"Mr. Ram is a great human being."* to

```
["Mr", " Ram is a great human being"]
```

As sentences may get arbitrarily split near the abbreviations, the context of a long sentence may get lost entirely. We can, of course, add more rules to our top-down approach, say, by looking for a space followed by a capital letter just after the end of a sentence. Though, the more rules we add, we can keep finding counter-examples where the defined set of rules do not hold. This is where a bottom-up approach might become useful where the system learns to segment sentences by itself, learning from relevant data in a particular domain.

Bottom-Up Approach - Punkt Sentence Tokenizer

The Punkt Sentence tokenizer is a bottom-up approach towards sentence segmentation which divides a text into a list of sentences by using an unsupervised algorithm to build a model for abbreviation words, collocations and words that start sentences. It can either be pre-trained on an extensive collection of plain-text in the target language or can be dynamically trained to learn parameters on incoming text.

The NLTK data package includes a pre-trained Punkt tokenizer for English. Though, when we are working on a specific target domain, the pre-trained models may be unsuitable. In such cases, we can use PunktTrainer to dynamically learn parameters such as a list of abbreviations from portions of text. Using a PunktTrainer directly allows for incremental training and modification of the hyper-parameters used to decide what is considered an abbreviation, etc. It is important to note that even though we call it unsupervised, there is inevitably a top-down invocation in the algorithm being chosen for learning.[1]

Naive Top-Down vs Pre-trained Punkt

- As evident, Punkt is much better at **handling abbreviations**. Since it has been pre-trained on multiple sentences, it has bottom-up knowledge of the structure of a sentence and therefore could recognize the fact that the *'dot'(.)* in the middle of a sentence is part of an abbreviation and does not denote the end of a sentence. For example, let us once again consider the text *"Mr. Ram is a great human being."*. The segments produced by the pre-trained Punkt tokenizer and our top-down naive approach are as follows:
  - **Punkt:** ["Mr. Ram is a great human being."]
  - **Naive:** ["Mr", " Ram is a great human being"]
- 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, if the query does not have space after the period, Punkt might treat them as abbreviations due to its bottom-up knowledge. This can lead to arbitrarily long sentences if the user is not careful enough. For example, let the text to be tokenized be *"Hello folks.Hope you all are having a great day.Welcome to the the greatest circus on earth."*. The segments produced by the pre-trained Punkt tokenizer and our top-down naive approach are as follows:
  - **Punkt:** ["Hello folks.Hope you all are having a great day.Welcome to the the greatest circus on earth."]
  - **Naive:** ["Hello folks", "Hope you all are having a great day", "Welcome to the greatest circus on earth"]

2 Word Tokenization

The most straightforward and apparent top-down approach to word tokenization is the use of spaces and symbols like hyphens(-), commas(,), forward-slashes(/), etc. to locate word endings. We have used the re.split function from the regular expression (re) python library to perform this naive approach.[2]

The Penn Treebank Word Tokenizer

The Penn Treebank tokenizer is a top-down approach for word tokenization which 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, therefore justifying it's top-down nature. It is important to note that Penn Treebank Tokenizer assumes that the text has already been segmented into sentences. The Penn Treebank tokenizer performs the following steps:

- Split standard contractions, e.g. "don't" to "do" and "n't"
- Treat most punctuation characters as separate tokens.
- Split off commas and single quotes when followed by a whitespace.
- Separate periods that appear at the end of line.

Naive Top-Down vs Penn Treebank

- The Penn Treebank Tokenizer is better at **handling standard contractions**. For example, let the sentence to be tokenized be *"I can't do this"*. The tokenized words produced by the Penn Treebank approach and our naive approach are as follows:
  - **Penn Treebank:** ["I", "ca", "n't", "do", "this"]
  - **Naive:** ["I", "can", "t", "do", "this"]

  Our naive approach tokenizes can't to 'can' and 't'. Hence, the information about negation (i.e., cannot) is lost. The Penn Treebank retains negation, as "n't" can be treated as "not" for many practical purposes.
- One area where our naive approach is clearly better than Penn Treebank is at **handling hyphens(-)**. The Penn Treebank ignores hyphens altogether. For example, let the sentence to be tokenized be *"The pre-civil war era in the US was a time of rapid agricultural expansion."*. The tokenized words produced by the Penn Treebank approach and our naive approach are as follows:
  - **Penn Treebank:** ["The", "pre-civil", "war", "era", "in", "the", "US", "was", "a", "time", "of", "rapid", "agricultural", "expansion"]
  - **Naive:** ["The", "pre", "civil", "war", "era", "in", "the", "US", "was", "a", "time", "of", "rapid", "agricultural", "expansion"]

  Words like "pre-civil" might not exist in the dictionary/corpora which might have further repercussions.

3 Stemming and Lemmatization

For grammatical reasons, documents use different forms of a word, such as organize, organizes, and organizing. Additionally, there are families of derivationally related words with similar meanings, such as democracy, democratic, and democratization. The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. Stemming refers to a crude heuristic process that chops off the ends of words in the hope of achieving the above goal and often includes the removal of derivational affixes. Lemmatization makes use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings and to return the base or dictionary form of a word known as the lemma.[3]

Major differences between Stemming and Lemmatization

If confronted with the token saw, stemming might return just s, whereas lemmatization would attempt to return either see or saw depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Stemmers use language-specific rules, but they require lesser knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to correctly lemmatize words.

4 Word Normalization for Information Retrieval

It is hard to clearly choose one over the other for IR systems as either form of normalization tends not to improve English information retrieval performance in aggregate - at least not by very much. While one or the other helps a lot for some queries, it equally hurts performance a lot for others. So the choice of Stemming or Lemmatization mainly depends on the use-case and what the user wants the IR system to do. There is an inherent Precision-Recall tradeoff that needs to be addressed. Search engines built with lemmatization are found to have higher precision, whereas, search engines built with Stemming give a much higher recall. For our application, we would like to design an IR system which provides the user with reasonably high number of choices and then let them decide which one they want to explore, i.e, we would like our system to have a higher recall even at the cost of lower precision. Hence, we have decided to use stemming. Apart from this, there is the apparent issue of latency which can be a pretty big bottleneck when the query sizes become large. Stemming is known to have a much lower latency due to its lower complexity.

5 Bottom-Up Approach for Stop-Word Removal

The most intuitive bottom-up approach for determining a stop list is to sort all the words in the corpus by collection frequency (the total number of times each word appears in the document collection), and then to take the most frequent words as stop words. Alternately, with some top-down invocation from Information Theory, stop words can be defined as the words which have the least discriminating ability, i.e., they carry very less information. The information contained (S) can be given as follows:

$$S = log(\frac{N}{n})$$

where,
N = Total number of documents in the corpus
n = Number of documents containing the given word

We can then choose the K least informative words as stop words. It is important to note that Web based search engines in general do not make use of stop words.

6 References

1. https://www.nltk.org/api/nltk.tokenize.html
2. https://docs.python.org/3/library/re.html
3. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
4. https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html
5. https://www.guru99.com/stemming-lemmatization-python-nltk.html
6. https://www.datacamp.com/community/tutorials/stemming-lemmatization-python