NLP Assignment - 1

Team Members:

- **Mohammed Safi Ur Rahman Khan (CS21M035)**
- **Ganesh Jatavath (CS21M019)**
- **Uday Sai Vemula (CS21M066)**

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

Answer:

Sentence segmentation is essentially the process of breaking down a given document/text into individual processing units (called sentences) consisting of typically more than one word.

The main task here is to identify the respective **sentence boundaries between words** in different sentences. In reality, there is no absolute definition for what constitutes a sentence. It is relatively arbitrary and greatly depends on the context and the language used. But anyways, sentences in most written languages are delimited by punctuation marks.

Therefore, recognizing boundaries in a written language involves determining the roles of all punctuation marks which can denote sentence boundaries. In the case of English, the typical sentence boundaries are **period(.)**, **question marks(?)**, **exclamation marks(!)**. Very **rarely**, semicolons(;), colons(:) and dashes(-) are also used.

In most NLP applications the only sentence boundary punctuation marks considered are the period, question marks, and exclamation marks. Hence for this purpose, we are going to stick with this.

The most simple and obvious top-down approach is to first find out what are the possible sentence boundaries (in our case we have already determined them to be period, question mark, and exclamation mark). Next, we go over our text and whenever we find the sentence boundary mark, we separate that sentence from the text and continue the process.

The simplest way to implement this is using a simple loop that iterates through the text. Whenever we find the sentence delimiter, we slice off the sentence from the text and append this in our list of sentences. The drawback of this method is that it is a bit lengthy to code and maybe inefficient (depending on the programming practices)

Another way to implement this is using Python's **split()** function. It takes in the sentence delimiter and automatically returns the list of sentences. The main drawback of this approach is that the function takes only a single delimiter. But in our case, we have 3 delimiters. So this method is not preferred.

The best way to implement this top-down approach is using regular expressions. We define a pattern that has the three delimiters i.e., '**[.?!]**'. Using this, we can use the **split()** function of **re** package and it returns us the list of sentences separated from the given text.

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 counterexample.

Answer:

No. The above-mentioned method **doesn't** always do the correct sentence segmentation. (We have stated the example to prove this later after giving some explanation).

In real-world scenarios, each of the above-used sentence delimiters can serve several different purposes. Some of these other purposes are listed below:

- A period is used to denote abbreviations also.
  Example: Mr. or St. or Ex. , etc
- A period is also used to denote decimal points.
  Example: 3.14
- A period can also be a part of “ellipses” i.e., ….
  Example: I was um… thinking about you today.
- Exclamation and question marks can also appear inside a quote.
  Example: “Why are you doing this?”, he asked.

These are some of the cases (there may be many more) where our sentence delimiters are used within the logical boundary of a sentence and hence our top-down approach fails.

Another case is that rarely sentences are delimited by other punctuation marks (as mentioned in our previous answer) also.
Example: Call me tomorrow; I can give you an answer then.

The article that we have referred to (check the references section) has outlined the various contextual factors that can influence the sentence segmentation process. They are

1. Case distinctions (not relevant for our corpus since everything is lower case)
2. Parts of speech - can help in disambiguating the punctuations.
3. Word length
4. Lexical endings - used to filter out words that weren't likely to be abbreviations.
5. Abbreviation classes.

One example of the sentence where our written code for the top-down approach fails is given below:

‘I need your name and address, please.’
‘I didn't really see anything.’
‘Your name and address!’ he said, abruptly. She immediately turned off the engine, startled.

This is an excerpt from the novel “Need you dead” written by Peter James.
As we can see, since the sentence has an exclamation mark in the quote, our approach fails to recognize this property and makes a mistake.

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?

Answer:

Punkt Sentence tokenizer essentially divides a given text into a list of sentences by using an unsupervised learning algorithm. This model accounts for abbreviations, collocations, and words that start sentences. It is typically trained on a large collection of plaintext in the language before being used.

In the top-down approach, we had kind of predefined rules for sentence segmentation. Therefore a fixed algorithm for all contexts. But, the algorithm even if it performs quite well on a certain corpus of data may not be successful for other corpora. Hence we need an approach to dynamically learn this segmentation process.

Punkt tokenizer is an implementation of the paper **“Unsupervised Multilingual Sentence Boundary Detection”** by Tibor Kiss and Jan Strunk. This paper proposes a language-independent, unsupervised approach for sentence boundary detection. It is based on the assumption that large ambiguities can be eliminated when the abbreviations have been determined. The system detects abbreviations using three criteria:

1. Abbreviations are defined as a very tight collocation consisting of a truncated word and a period.
2. They are usually short.
3. They can sometimes contain internal periods.

After this task of abbreviation detection is done, as the paper shows, the majority of ambiguities are removed from the task of sentence boundary detection.
Quoting the authors of the paper “*Quantitatively, abbreviations are a major source of ambiguities in sentence boundary detection since they often constitute up to 30% of the possible candidates for sentence boundaries in running text*”
and
“*The determination of abbreviation types already yields a large percentage of all sentence boundaries because all periods occurring after non-abbreviation types can be classified as end-of-sentence markers.”*

This method doesn't make use of additional annotations, POS tagging, or pre-compiled lists to support the sentence boundary detection, but rather extracts all the necessary data from the training corpus.

Hence, this is a bottom-up approach wherein the rules and other stuff are actually learned from the data and not prespecified.

4. Perform sentence segmentation on the documents in the Cranfield dataset using:
(a) The top-down method stated above
(b) The pre-trained Punkt Tokenizer for English
State a possible scenario along with an example where:
(a) the first method performs better than the second one (if any)
(b) the second method performs better than the first one (if any)

Answer:

The code is written and submitted in python. For the top-down approach, we have used the regular expression method, wherein we defined the pattern as '**[.?!]**' .

For the bottom-up approach using Punkt tokenizer, we had two options. One was directly using **sent_tokenize()** function implemented in the **nltk.tokenize** package. According to the documentation, (link attached in the references below), it uses the NLTK's recommended sentence tokenizer which is currently **PunktSentenceTokenizer.**

Another approach was a bit lengthy method (but is more customizable). We load the tokenizer as a pickle file from **'tokenizers/punkt/english.pickle'** using **nltk.data.load()** function. Using this loaded tokenizer, we run the **.tokenize()** method to return a list of sentences from the given text.

Scenarios where the methods perform better than the other:

(a) The only scenario where I can think that this method performs better than the other method is in terms of time and space. For the bottom-up method, (if say we were not to use the libraries), we would have needed to train the algorithm on the data for our current domain which needs a lot of time as well as space. This is a complex method that also requires a lot of data.

Also, since this is an unsupervised learning method, a **very remote possibility** is there wherein our model learns the wrong things and **MAY** (enhanced emphasis on “may”) do the sentence segmentation incorrectly.

(b) There are many scenarios where the bottom-up approach performs better than the top-down approach. (As seen earlier)

For the sake of completeness, quoting a statement from the article (referenced below) where the top-down approach fails but the bottom-up approach performs much better.

“There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit say to itself, ‘Oh dear! Oh dear! I shall be late!’ (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOATPOCKET, and looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it, and fortunately was just in time to see it pop down a large rabbit-hole under the hedge”

(A line from Alice in Wonderland)

This example contains a single period at the end and three exclamation points within a quoted passage.
Our top-down approach gives us 3 sentences for the above text. (wrong)
Our bottom-up approach gives us just 1 sentence for the above text (assuming that semicolon is not considered to be the end of sentence) which is correct.

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

Answer:

Before we get into the details of the simplest top-down approach to word tokenization for English texts. Let us understand what tokenization is: Tokenization is the process of breaking up the existing sequence of the characters in a text by locating the word boundaries.In other words, the exact point where one word ends and another one begins.

In the case of natural languages, the word boundaries are not well-established and well-defined. It is very different and ambiguous when we go from one language to another. But when it comes to artificial languages, like any programming language, tokenization is very well defined, established, and well-understood, Same not in the case of natural languages like English, Spanish, and others. Due to tokenization rules not being strict and well-defined it is very difficult to eliminate lexica & structural ambiguities.

In natural language the same character can serve many different purposes and in which syntax is not strictly defined. There are multiple factors which can potentially affect the difficulty in tokenizing any natural language. Broadly we can observe one fundamental difference between the approaches for space-delimited languages and approaches for unsegmented languages:

There exist several approaches and methods to word tokenize for english text:

1. One among them is Breaking the given string at each space because In many alphabetic writing systems. Including those that uses alphabets, words are

separated by spaces, english is a space-delimited language so it suits well. Above approach can be implemented by the split() function.

2. The most tokenization ambiguity exist among uses of punctuation marks such as
    1. (.)period
    2. (,)commas
    3. (“”)quotations
    4. Apostrophes and hyphens

Since the same punctuation mark can serve many different functions in a single sentence.

A logical initial tokenization of a space-delimited language would consider as a separate token any sequence of characters preceded and followed by space.

This successfully tokenizes words that are a sequence of alphabetic characters but does not take into account punctuation characters.

Different Methods to Perform Tokenization in Python

- Tokenization using Python split() Function
- Tokenization using Regular Expressions
- Tokenization using NLTK
- Tokenization using Spacy
- Tokenization using Keras
- Tokenization using Gensim

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

Answer:

The treebank tokenizer uses regular expressions to tokenize the text in Penn treebank. It assumes that text has already been split into sentences which can be done by the **sent_tokenize()** function in nltk library.

Like in artificial languages tokenization is well-established and well-understood, the same is not the case in natural language tokenization. So for artificial languages, it is easy to eliminate the lexical and structural ambiguities.

Since natural languages are not strictly defined syntactically it is very difficult to tokenize without additional information about the languages like: additional lexical and morphological information.

This tokenizer looks to be top-down knowledge-based as it is using pre-defined regular expression rules to perform tokenization.

NLTK's Penn Treebank tokenizer performs the following tasks:

1. Split the standard contraction eg: **they'll**->**they'll**, **she's-> she's**
2. Treat most punctuation characters as separate tokens.
3. Split off **(,)comma** and **single quote**, when followed by whitespace.
4. Separate periods that appear at the end of the line.

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

(a) The simple method stated above

(b) Penn Treebank Tokenizer

State a possible scenario along with an example where:

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

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

Answer:

The naive tokenizer function is implemented using **split()** function of **re** and Penn Treebank Tokenizer implemented by using Treebank Tokenizer of NLTK and python code is added.

We have used a simple approach to tokenize the given text. i.e, Tokenization using python's split() function. Returns a list of strings after breaking the given string by specified separator. By default split() breaks a string at each space. We can change the separator to anything.

a. The biggest drawback of using **split()** function for tokenizing is that it allows to use only one separator at a time. And the **split()** function does not consider **punctuation** as

a separate token. As we can easily guess, it is not an efficient approach to tokenize the text.

This approach might work better when we tokenize the given string based on one separator like white space and in special cases for **example: can't**->treebank will split it as **ca** and **n't**. **ca** has no meaning in the English dictionary. A naive approach divides the whole as one token i.e, **can't** so, it will perform better in these special cases.

b. Yes, The second approach performs better than the first approach. We have used NLTK's Penn Treebank tokenizer as a second method to perform the tokenization and it performs better than the naive or first approach which we used in tokenization.

NLTK's Penn Treebank tokenizer considers the punctuation as a separate token. And it takes many other cases like:

Splitting the standard contraction, for example: **he's** -> **he** and **'s**, it separates periods that appear **at the end of the line,** Split off **(,)comma and single quote**, when followed by **whitespace**. NLTK's Penn Treebank tokenizer is very fast compared to the first approach.

8. What is the difference between stemming and lemmatization?

Stemming just removes or stems the last few characters of a word without considering the context of that particular word, often leads to a different form which is not identical to morphological root of that word. Often the words doesn't exist in the dictionary of that particular language.

Lemmatization can be considered as more sophisticated form of stemming which is aware of context of the word, converts into meaningful base form which is called lemma. Lemmatization takes into consideration the morphological analysis of the word. It converts into meaningful base form so it needs a lookup into dictionaries.

Often it is a trade-off between speed and accuracy of the model. Stemming is faster than lemmatization, stemming works with a set of rules to look into, whereas lemmatization needs more computation to consider parts of speech and valid dictionary words etc. recall(fraction of relevant instances that were retrieved) is almost always higher in lemmatization than stemming because of its morphological analysis of the word.

Original	**Stemming**	**Lemmatization**
Advisable	Advis	Advise
Studies	studi	study
studying	study	study
United	Unite	United
Caring	car	care
believes	believ	belief
పంచను (I don’t share)	పంచ (loincloth)	పంచు (share)

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

We decided to use lemmatization instead of stemming in our search engine application, the following reasons can quantify the decision,

- Lemmatization is more sophisticated than stemming.
- Recall is more with the use of lemmatization than stemming, in search engine application it is often cherishing to find the most relevant document than finding the not so relevant document faster.
- In the given data we only have around 1400 documents to search queries on, which can be considered as a small dataset so experimenting on lemmatization to see the relevance in documents is better since we are not falling short on computation power as well.
- Probably for the vast dataset, stemming is preferred in search engine applications because we never display our stems of words to the users they only processed internally and stemming can be used only when speed of the model matters a lot.
- Most often stemming shows the straightforward trimmed words which might re-define the meaning of sentence altogether. The following can be a best example

Ex:- the word ‘caring’ will be converted to ‘car’ after stemming, which has totally different meaning, whereas lemmatization gives the correct morpheme which is ‘care’.

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

We are using WordNetLemmatizer provided by nltk libraries. It takes two arguments majorly, one is the token that should be lemmatized and the other one is its parts of

speech (often referred as pos). so before lemmatizing the tokens we need to figure out their parts of speech, for that we are using **pos_tag** function provided by nltk itself, pos_tag() takes list of sentences where each sentence has a list of words and gives back tuple pairs of words and their parts of speech. Pos_tag() has very rich division parts of speech but we are restricting them to 4 major parts to lemmatize, which are NOUN, ADJECTIVE, VERB, ADVERB.

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

Answer:

We loaded the list of stopwords from **nltk.corpus** package.
Then we iterate through all the lists. In each list, we compare each token and see whether it is present in the stopwords list or not. If it is present, then we remove it from the list.
We return the final list of list of tokens with stopwords removed.

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

Answer:

Any part of natural language text contains seemingly contentless material (or so we think sometimes) called as stop words, which doesn't really add any value in our complex models but just seems to hog the space and resources. That is why often it is better to remove such stop words.
Many NLP libraries come equipped with lists of stop words, but there is no fixed definition of a stop word. As said, this approach of using a predefined list of words for stop word removal is a top-down approach. But here’s the catch: there’s no universal

stop words list because a word can be empty of meaning depending on the corpus you are using or on the problem you are analyzing. This means that any word can be a stop word depending on what you are trying to do.

For seeing a possible bottom-up approach, we need to first see what words are classified as stop words. Often frequently occurring words in the corpus are treated as the stop words (for that particular corpus. May not be true for other corpora). Rules of thumb like selecting the 10-100 most frequent words in a body of text are also common ways of identifying stop words.

In essence, we go through the corpus and evaluate the frequency of each word appearing in the corpus. Then we can fix a threshold and treat those words that have a frequency greater than the threshold as stopwords.
(Inverse Document Frequency (IDF) of each word is actually better than Term Frequency (TF) as TF is adversely affected by a few documents that contain a word many times)
This threshold is not fixed. It varies from corpus to corpus and can be estimated by seeing the distribution of the frequencies and using domain knowledge.

We can enhance this method by also including POS tagging and thereby removing words like articles, some kinds of proper nouns, etc.

We can also combine both the top-down and bottom-up approaches. I.e., Also maintain a predefined list of stopwords along with learning the stopwords from the data.

References:

- https://tm-town-nlp-resources.s3.amazonaws.com/ch2.pdf
- https://www.nltk.org/api/nltk.tokenize.html
- https://www.analyticsvidhya.com/blog/2019/07/how-get-started-nlp-6-unique-ways-perform-tokenization/

- https://www.nltk.org/_modules/nltk/tokenize/punkt.html
- https://aclanthology.org/J06-4003.pdf
- https://www.nbccomedyplayground.com/what-is-penn-treebank-tokenizer/
- https://www.nltk.org/_modules/nltk/tokenize/treebank.html
- https://www.analyticsvidhya.com/blog/2019/07/how-get-started-nlp-6-unique-ways-perform-tokenization/
- https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
- https://www.baeldung.com/cs/stemming-vs-lemmatization
- https://www.analyticsvidhya.com/blog/2021/11/an-introduction-to-stemming-in-natural-language-processing/
- https://www.tutorialspoint.com/natural_language_toolkit/natural_language_toolkit_stemming_lemmatization.htm#:~:text=Stemming%20is%20a%20technique%20used,stemming%20for%20indexing%20the%20words.
- https://medium.com/@limavallantin/why-is-removing-stop-words-not-always-a-good-idea-c8d35bd77214
- https://www.geeksforgeeks.org/removing-stop-words-nltk-python/
- https://smltar.com/stopwords.html