NLP Text Preprocessing: A Practical Guide and Template

Jiahao Weng Aug 30, 2019 · 6 min read ★

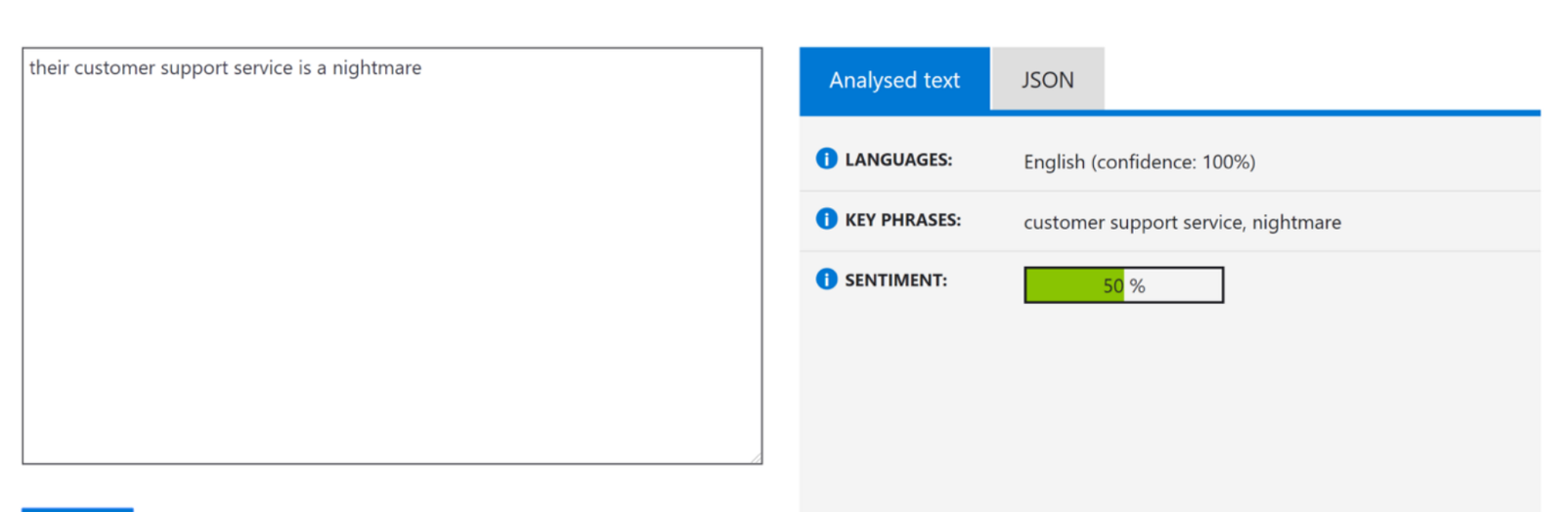
Text preprocessing is traditionally an important step for natural language processing (NLP) tasks. It transforms text into a more digestible form so that machine learning algorithms can perform better.

Importance of Text Preprocessing

To illustrate the importance of text preprocessing, let's consider a task on sentiment analysis for customer reviews.

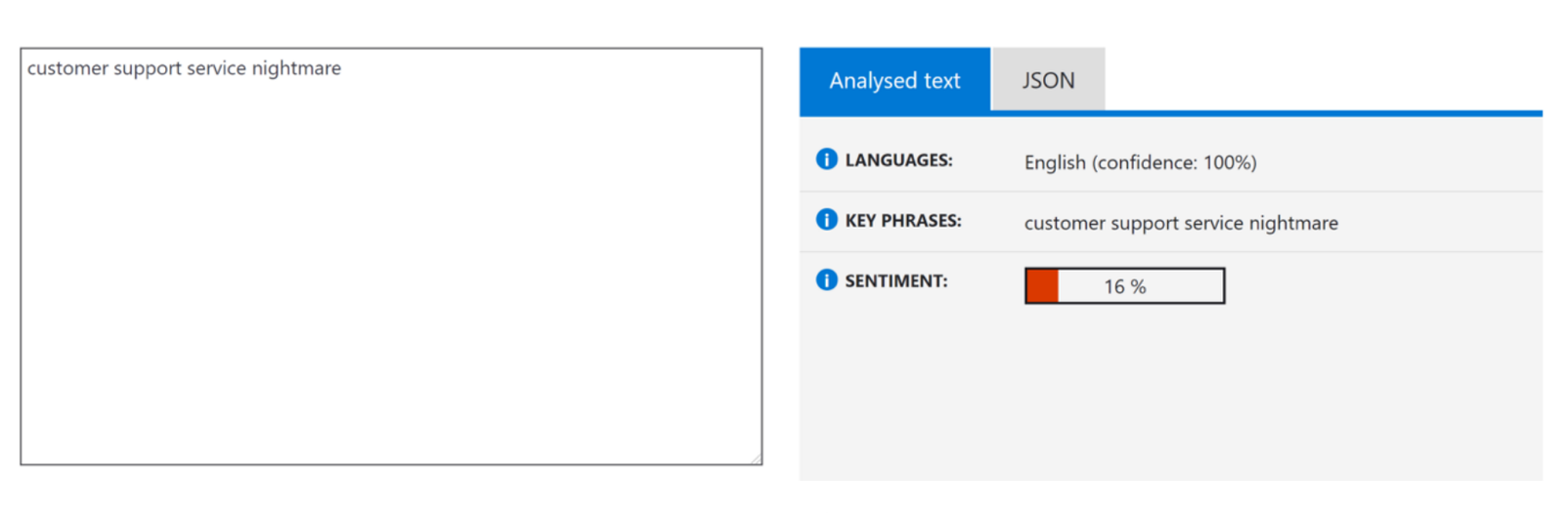
Suppose a customer feedbacked that "their customer support service is a nightmare", a human can surely and clearly identify the sentiment of the review as negative. However for a machine, it is not that straightforward.

To illustrate this point, I experimented with the Azure text analytics API. Feeding in the same review, the API returns a result of 50%, i.e., neutral sentiment, which is wrong.



Azure text analytics API results as of publication date (30 Aug 2019)

However, if we had performed some text preprocessing, in this case just removing some stopwords (explained further below but for now, think of stopwords as very common words such that they do not help much in our NLP tasks), we will see that the results become 16%, i.e., negative sentiment, which is correct.



Azure text analytics API results after our own preprocessing

So as illustrated, text preprocessing if done correctly can help to increase the accuracy of the NLP tasks.

Side note: The above Azure example is actually even more interesting because by right, Azure's text analytics API should have already processed the text as part of its model, but somehow it seems that the stopwords are confounding its model.

General Outline of Text Preprocessing

So how do we go about doing text preprocessing? Generally, there are 3 main components:

- Tokenization
- Normalization
- Noise removal

In a nutshell, **tokenization** is about splitting strings of text into smaller pieces, or "tokens". Paragraphs can be tokenized into sentences and sentences can be tokenized into words. **Normalization** aims to put all text on a level playing field, e.g., converting all characters to lowercase. **Noise removal** cleans up the text, e.g., remove extra whitespaces.

For details, please refer to this great article by Matthew Mayo.

List of Text Preprocessing Steps

Based on the general outline above, we performed a series of steps under each component.

1. Remove HTML tags
2. Remove extra whitespaces
3. Convert accented characters to ASCII characters
4. Expand contractions
5. Remove special characters
6. Lowercase all texts
7. Convert number words to numeric form
8. Remove numbers
9. Remove stopwords
10. Lemmatization

Link to full code can be found at bottom of article, but read on to understand the salient steps taken.

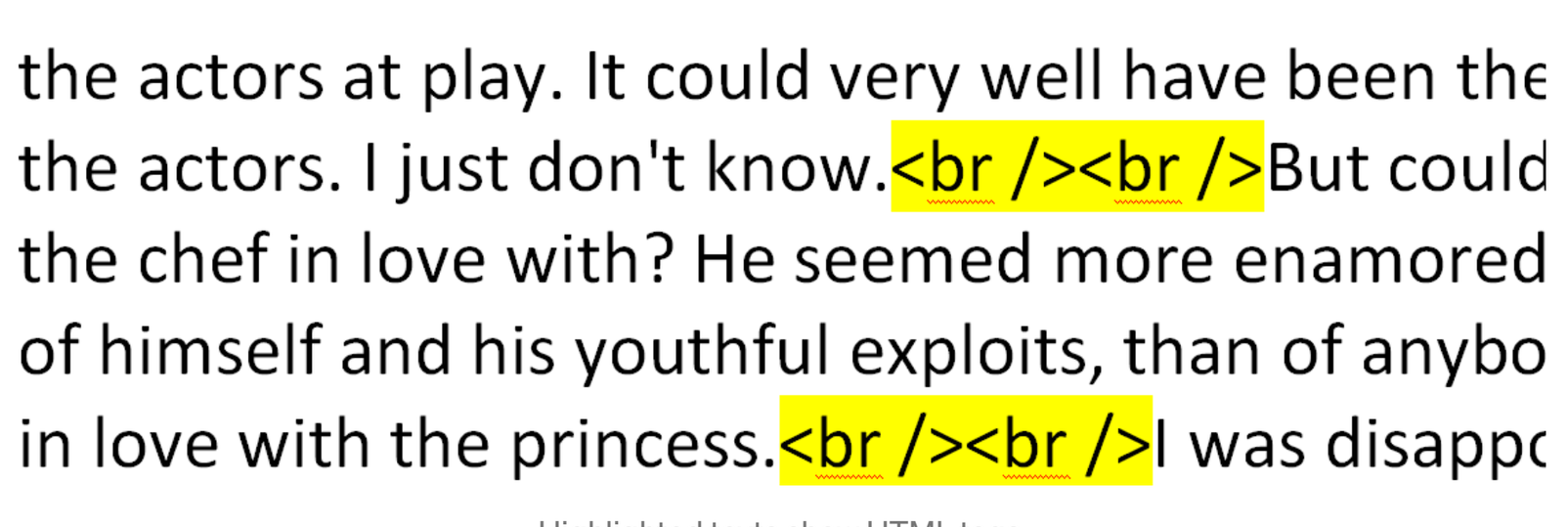
The necessary dependencies are as such:

```
from bs4 import BeautifulSoup
import spacy
import unidecode
from word2number import w2n
import contractions

# load spacy model, can be "en_core_web_sm" as well
nlp = spacy.load('en_core_web_md')
```

Remove HTML Tags

If the reviews or texts are web scraped, chances are they will contain some HTML tags. Since these tags are not useful for our NLP tasks, it is better to remove them.



Highlighted texts show HTML tags

To do so, we can use BeautifulSoup's HTML parser as follows:

```
def strip_html_tags(text):
    """remove html tags from text"""
    soup = BeautifulSoup(text, "html.parser")
    stripped_text = soup.get_text(separator=" ")
    return stripped_text
```

Convert Accented Characters

"Would you like to have latté at our café?"

Words with accent marks like "latté" and "café" can be converted and standardized to just "latte" and "cafe", else our NLP model will treat "latté" and "latte" as different words even though they are referring to same thing. To do this, we use the module unidecode.

```
def remove_accented_chars(text):
    """remove accented characters from text, e.g. café"""
    text = unidecode.unidecode(text)
    return text
```

Expand Contractions

Contractions are shortened words, e.g., don't and can't. Expanding such words to "do not" and "can not" helps to standardize text.

We use the contractions module to expand the contractions.

```
def expand_contractions(text):
    """expand shortened words, e.g. don't to do not"""
    text = contractions.fix(text)
    return text
```

Note: This step is optional depending on your NLP task as spaCy's tokenization and lemmatization functions will perform the same effect to expand contractions such as can't and don't. The slight difference is that spaCy will expand "we're" to "we be" while pycontractions will give result "we are".

Treatment for Numbers

There are two steps in our treatment of numbers.

One of the steps involve the conversion of number words to numeric form, e.g., seven to 7, to standardize text. To do this, we use the word2number module. Sample code as follows:

```
text = """three cups of coffee"""

doc = nlp(text)

tokens = [w2n.word_to_num(token.text) if token.pos_ == 'NUM' else
token for token in doc]

print(tokens) # result: [3, cups, of, coffee]
```

The other step is to remove numbers. As you shall see later, we are able to toggle on or off the steps by setting parameters to *True* or *False* value. Removing numbers may make sense for sentiment analysis since numbers contain no information about sentiments. However, if our NLP task is to extract the number of tickets ordered in a message to our chatbot, we will definitely not want to remove numbers.

Stopwords

As mentioned earlier, stopwords are very common words. Words like "we" and "are" probably do not help at all in NLP tasks such as sentiment analysis or text classifications. Hence, we can remove stopwords to save computing time and efforts in processing large volumes of text.

In our case, we used spaCy's inbuilt stopwords, but we should be cautious and modify the stopwords list accordingly. E.g., for sentiment analysis, the word "not" is important in the meaning of a text such as "not good". However, spaCy included "not" as a stopword. We therefore modify the stopwords by the following code:

```
# exclude words from spacy stopwords list
deselect_stop_words = ['no', 'not']
for w in deselect_stop_words:
    nlp.vocab[w].is_stop = False
```

Lemmatization

Lemmatization is the process of converting a word to its base form, e.g., "caring" to "care". We use spaCy's lemmatizer to obtain the lemma, or base form, of the words. Sample code:

```
text = """he kept eating while we are talking"""
doc = nlp(text)

# Lemmatizing each token
mytokens = [word.lemma_ if word.lemma_ != "-PRON-" else word.lower_
for word in doc]

print(mytokens)
# result: ['he', 'keep', 'eat', 'while', 'we', 'be', 'talk']
```

Another method to obtain the base form of a word is stemming. We did not use it in our text preprocessing code but you can consider stemming if processing speed is of utmost concern. But do take note that stemming is a crude heuristic that chops the ends off of words and hence, the result may not be good or actual words. E.g., stemming "caring" will result in "car".

Everything Together Now and Next Steps

Putting everything together, the full text preprocessing code is as such:

https://gist.github.com/jiahao87/d57a2535c2ed7315390920ea9296d79f

Sample code to run the function is as follows:

```
# text = """I'd like to have three cups   of coffee<br /><br />from
your Café. #delicious"""

text_preprocessing(text)
# result: ['like', 'cup', 'coffee', 'cafe', 'delicious']
```

To toggle on or off specific steps, we can set the relevant parameters to *True* or *False* value. E.g., to not remove numbers, set the parameter "remove_num" to *False*.

```
# example to not remove numbers
text_preprocessing(text, remove_num=False)
```

After this, we can then convert the processed text into something that can be represented numerically. Two main ways of doing so are one-hot encodings and word embedding vectors. We shall explore these in the next article.

Lastly, do note that there are experts who expressed views that text preprocessing negatively impact rather than enhance the performance of deep learning models. Nonetheless, text preprocessing is definitely crucial for non-deep learning models.

Thanks for reading and I hope the code and article are useful. Please also feel free to comment with any questions or suggestions you may have.

References

- https://www.kdnuggets.com/2017/12/general-approach-preprocessing-text-data.html
- https://www.kdnuggets.com/2018/08/practitioners-guide-processing-understanding-text-2.html
- https://docs.microsoft.com/en-in/azure/cognitive-services/text-analytics/how-tos/text-analytics-how-to-sentiment-analysis
- https://pypi.org/project/contractions/
- https://pypi.org/project/word2number/