Various text pre-processing techniques in NLP

Archana Devi Ramesh

I. Introduction

Natural Language Processing (NLP) is a branch of Artificial Intelligence that analyzes, processes, and efficiently retrieves information text data. By utilizing the power of NLP, one can solve a huge range of real-world problems which include summarizing documents, title generator, caption generator, fraud detection, speech recognition, recommendation system, machine translation, etc.

In any such task, cleaning or pre-processing the data is as important as model building. Text data is one of the most unstructured forms of available data and when comes to deal with human language then it's becomes too complex.Text pre-processing is a method to clean the text data and make it ready to feed data to the model. Text data contains noise in various forms like emotions, punctuation, text in a different case. In human language then, there are different ways to say the same thing, And this is only the main problem we have to deal with because machines will not understand words, they need numbers so we need to convert text to numbers in an efficient manner. This plays a major role in deciding the performance of the model because the performance of the model is based on how well the raw data is cleaned and pre-processed. This article will go through the main text pre-processing techniques that one must know to work with any textual data.

II. Libraries used

There are many libraries and algorithms used to deal with NLP-based problems. A regular expression(re) is mostly used library for text cleaning. NLTK (Natural language toolkit) and spacy are the next level library used for performing Natural language tasks like removing stopwords, named entity recognition, part of speech tagging, phrase matching, etc.

Run the following command in the IDE inorder to install nltk.

```
pip install nltk
```

III. Dataset exploration

In this article I will be considering the UCI SMS spam classification dataset for the practical implementation. Let us first explore this dataset.

```
import pandas as pd
df = pd.read_csv('SMSSpamCollection', sep='\t', names = ['label', 'message'])
df.head()
```

The first 5 entries in the dataset looks like the following. The dataset contains a **label** column which says the message as ham or spam. The next columns **message** contains the sms message to be classified as ham or spam.

	label	message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Fig. 1. Reading the dataset

Dwelling into the first 5 messages, we can observe lots of noise like punctuation marks, special characters, extra spaces, different cases and many more. .

```
for mess in df['message'].head():
   print('\n',mess)
```

```
Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...

Ok lar... Joking wif u oni...

Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&
C apply 08452810075over18

U dun say so early hor... U c already then say...

Nah I do not think he goes to usf, he lives around here though
```

Fig. 2. First 5 SMS

IV. Various pre-processing techniques

In this section various text pre-processing techniques like expanding contractions, converting to lower case, punctuation removal, digits removal, stop words removal, stemming and lemmatization are covered with pratical implementation. The entire code for this implementation can be found at Github repo.

A. Expanding contractions

Contraction is the shortened form of a word like **don't** stands for **do not**, **aren't** stands for **are not**. Like this, we need to expand this contraction in the text data for better analysis. you can easily get the dictionary of contractions on google or create your own and use the regular expression module to map the contractions.

```
contractions = {
"ain't": "am not","aren't": "are not","can't": "cannot","can't've": "cannot have","'cause":
    "because","could've": "could have","couldn't": "could not",
"couldn't've": "could not have","didn't": "did not","doesn't": "does not","don't": "do
    not","hadn't": "had not","hadn't've": "had not have",
"hasn't": "has not","haven't": "have not","he'd": "he would","he'd've": "he would
    have","he'll": "he will","he's": "he is","how'd": "how did",
"how'll": "how will","how's": "how is","i'd": "i would","i'll": "i will","i'm": "i am","i've":
    "i have","isn't": "is not","it'd": "it would",
"it'll": "it will","it's": "it is","let's": "let us","ma'am": "madam","mayn't": "may
    not","might've": "might have","mightn't": "might not",
"must've": "must have","mustn't": "must not","needn't": "need not","oughtn't": "ought
    not","shan't": "shall not","sha'n't": "shall not","she'd": "she would",
"she'll": "she will","she's": "she is","should've": "should have","shouldn't": "should
    not","that'd": "that would","that's": "that is","there'd": "there had",
"there's": "there is","they'd": "they would","they'll": "they will","they're": "they
    are","they've": "they have","wasn't": "was not","we'd": "we would",
"we'll": "we will","we're": "we are","we've": "we have","weren't": "were not","what'll": "what
    will","what're": "what are","what's": "what is",
"what've": "what have","where'd": "where did","where's": "where is","who'll": "who
    will","who's": "who is","won't": "will not","wouldn't": "would not",
"you'd": "you would","you'll": "you will","you're": "you are"
}

def contract(text):
   new_text = ""
   i = 0
   for word in text.split():
      if i==0:
         new_text = contractions.get(word,word)
      else:
         new_text = new_text + " " + contractions.get(word,word)
      i += 1
   return new_text.replace("'s", '')

df['message'] = df['message'].apply(lambda x: contract(x))
df.head()
```

The above code segment expands all contractions, let us check the message in line number 5 which previously had a contraction. It is evident that **don't** is being converted to **do not**.

```
df['message'][4]
```

```
'Nah I do not think he goes to usf, he lives around here though'
```

Fig. 3. After expanding contraction

B. Lower casing

As the name implies, in this section we'll convert our text data into lower case. For a text input, such as paragraph there will be words in both lower and upper case. However, the computer considers the words written in different cases as different entities. For example, **Hello** and **hello** are considered as two different words by the computer though they are the same word. In order to avoid this type of controversy, we must convert all the words to lower case. There is a **lower()** method in python that takes care of this functionality. The below code segment converts all the characters to lower case.

```
import string
df['message'] = df['message'].str.lower()
df.head()
```

	label	message
0	ham	go until jurong point, crazy.. available only ...
1	ham	ok lar... joking wif u oni...
2	spam	free entry in 2 a wkly comp to win fa cup fina...
3	ham	u dun say so early hor... u c already then say...
4	ham	nah i don't think he goes to usf, he lives aro...

Fig. 4. After converting to lower case

C. Removing punctuations

Removing punctuations from the text is the most common text processing technique. The removal of punctuation helps to treat all text equally. For example, **Wow** and **Wow!** are treated equally. There are a total 32 main punctuations that need to be taken care of. We can directly use the string module with a regular expression to replace any punctuation in text with an empty string. 32 punctuations which string module provide us is listed below.

```
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

The code to remove punctuations using the python module **string.punctuation** is shown below. In each of the sms, it checks each and every character. If it does not belong to a punctuation, then it is stored in a list using list comprehension. In the end, the characters and joined in the list as a string and returned.

```
def rem_punct(mess):
    nopunc = ''.join([c for c in mess if c not in string.punctuation])
    return nopunc

df['message'] = df['message'].apply(rem_punct)
```

The output afterwards is the following figure.

	label	message
0	ham	go until jurong point crazy available only in ...
1	ham	ok lar joking wif u oni
2	spam	free entry in 2 a wkly comp to win fa cup fina...
3	ham	u dun say so early hor u c already then say
4	ham	nah i do not think he goes to usf he lives aro...

Fig. 5. After removing punctuations

D. Removing digits

Sometimes it happens that words and digits are written combined in the text which creates a problem for machines to understand. Hence, We need to remove such words which are combined like b4 or 2morrow. This type of word is difficult to process so better to remove them or replace them with an empty string.

Consider the sms given below. It has numbers and words containing numbers.

```
df['message'][2]
```

```
'free entry in 2 a wkly comp to win fa cup final tkts 21st may 2005 text fa to 87121 to receive entry questionstd txt ratetc ap
ply 08452810075over18'
```

Fig. 6. Before removing digits

In order to remove digits, the python module **isdigit()** can be used. The below code segment uses list comprehension in a similar way the punctuations were removed.

```
def rem_digit(mess):
    nodigit = ''.join([c for c in mess if not c.isdigit()])
    return nodigit

df['message'] = df['message'].apply(rem_digit)
```

After removal, the sms looks like the following.

```
df['message'][2]
```

```
'free entry in  a wkly comp to win fa cup final tkts st may  text fa to  to receive entry questionstd txt ratetc apply over'
```

Fig. 7. After removing digits

E. Stop words removal

Stopwords are the most commonly occurring words in a text which do not provide any valuable information. Words like **they, there, this, where,** etc are some of the stopwords. NLTK library is a common library that is used to remove stopwords and include approximately 180 stopwords which it removes. If we want to add any new word to a set of words then it is easy using the add method. For that first import the nltk module and download the corpus of stopwords and also import tokenize to convert the sentences into tokens.

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
from nltk.tokenize import word_tokenize
```

To remove stopwords, the below code can be run.

```
def remove_stop(mess):
    word_tokens = word_tokenize(mess)
    nostop = [word for word in word_tokens if not word in stopwords.words('english')]
    return nostop

df['message'] = df['message'].apply(remove_stop)
```

After removing stopwords, the output looks like the following. We can observe that the stopwords like **so, I, he** etc are removed comparing Fig 1.

	label	message
0	ham	[go, jurong, point, crazy, available, bugis, n...
1	ham	[ok, lar, joking, wif, u, oni]
2	spam	[free, entry, wkly, comp, win, fa, cup, final,...
3	ham	[u, dun, say, early, hor, u, c, already, say]
4	ham	[nah, think, goes, usf, lives, around, though]

Fig. 8. After removing stopwords

F. Stemming

Stemming is the process of reduction of a word to its root or stem word. The root form is left behind by removing the word affixes. For example, the words **plays**, **playing**, **played** are all reduced to its root word **play**. There are various types of stemming algorithms like porter stemmer, snowball stemmer. Porter stemmer is widely used present in the NLTK library.

In the below code, using PorterStemmer, the function **stem_words** takes each of the sms, and does stemming of each word in it.

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
def stem_words(text):
    return [stemmer.stem(word) for word in text]

df['message1'] = df['message'].apply(lambda x: stem_words(x))
```

After stemming, the output looks like the following. We can see that words like **joking** gets converted to **joke** and **lives** to **live**. we can see that some of the words it has stemmed which is not required. Stemming does not always result in words that are part of the language vocabulary. It often produces words with no meaning. That's only the disadvantage of this.

	label	message	message1
0	ham	[go, jurong, point, crazy, available, bugis, n...	[go, jurong, point, crazi, avail, bugi, n, gre...
1	ham	[ok, lar, joking, wif, u, oni]	[ok, lar, joke, wif, u, oni]
2	spam	[free, entry, wkly, comp, win, fa, cup, final,...	[free, entri, wkli, comp, win, fa, cup, final,...
3	ham	[u, dun, say, early, hor, u, c, already, say]	[u, dun, say, earli, hor, u, c, alreadi, say]
4	ham	[nah, think, goes, usf, lives, around, though]	[nah, think, goe, usf, live, around, though]

Fig. 9. After stemming

G. Lemmatization

Lemmatization is similar to stemming, used to stem the words into root word but differs in working. Actually, Lemmatization is a systematic way to reduce the words into their lemma by matching them with a language dictionary. That is Lemmatization is the process of converting the words in a text into a meaningful parent word.

The below code is used to perform lemmatization and it uses the nltk module **WordNetLemmatizer**.

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
def lemmatize_words(text):
   return [lemmatizer.lemmatize(word) for word in text]

df['message2'] = df['message'].apply(lambda text: lemmatize_words(text))
```

In the below Figure, the output both after stemming and lemmatization can be seen and it can be observed that lemmatization does not produce a non meaningful word like stemming.

```
df.head()
```

	label	message	message1	message2
0	ham	[go, jurong, point, crazy, available, bugis, n...	[go, jurong, point, crazi, avail, bugi, n, gre...	[go, jurong, point, crazy, available, bugis, n...
1	ham	[ok, lar, joking, wif, u, oni]	[ok, lar, joke, wif, u, oni]	[ok, lar, joking, wif, u, oni]
2	spam	[free, entry, wkly, comp, win, fa, cup, final,...	[free, entri, wkli, comp, win, fa, cup, final,...	[free, entry, wkly, comp, win, fa, cup, final,...
3	ham	[u, dun, say, early, hor, u, c, already, say]	[u, dun, say, earli, hor, u, c, alreadi, say]	[u, dun, say, early, hor, u, c, already, say]
4	ham	[nah, think, goes, usf, lives, around, though]	[nah, think, goe, usf, live, around, though]	[nah, think, go, usf, life, around, though]

Fig. 10. After lemmatization

V. Conclusion

In this article, I have covered the most essential and important text pre-processing techniques in Natural Language Processing with hands-on practice. You can also develop a single function with all techniques to pre-process text. These are very useful to improve the performance of the model by effectively cleaning and processing the data.

References

[1] https://medium.com/mlearning-ai/important-text-pre-processing-techniques-for-nlp-ea7d707e0e15
[2] https://www.analyticsvidhya.com/blog/2021/06/must-known-techniques-for-text-preprocessing-in-nlp/
[3] https://towardsdatascience.com/text-preprocessing-in-natural-language-processing-using-python-6113ff5decd8