TEXT PREPROCESSING FOR SPAM FILTERING

*“Garbage in Garbage Out”* this quote is one of the key fundamentals when it comes to machine learning and Natural language processing. Natural language processing is no different, it is a stream of data science where natural language data is given to machine learning algorithms for various purposes like: -

1. Sentiment Analysis
2. Spam filtering
3. Named entity recognition
4. Part of speech tagging etc.

In this blog I will be covering some very useful text pre-processing techniques and we will see how text pre-processing can make a difference in your final results.

Some pre-processing steps that we will cover in this blog will be :-

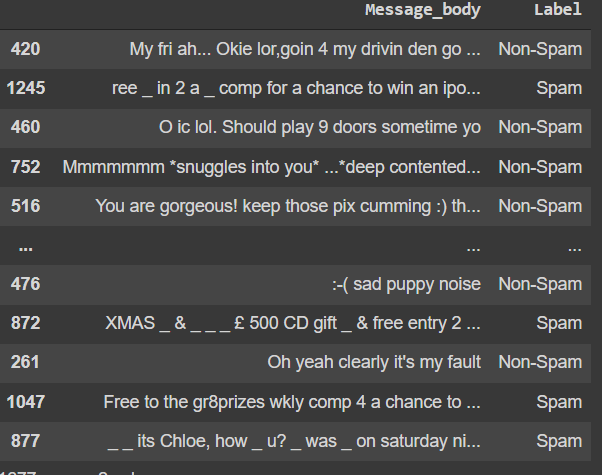
1. Lower casing
2. Removal of punctuations
3. Removal of stop words
4. Removal of frequent words
5. Stemming
6. Lemmatization
7. Conversion of Emoticons to words
8. Removal of URLs
9. Removal of HTML tags
10. Spelling Correction
11. Removal of rare words

Remember we do not have to use every text pre-processing technique for our data and for every different use case we have to carefully select which works best for us.

For example, In case of sentiment analysis it is better not to remove emoticons since emoji convey some very important information to our data but it is useful to convert the emoticon back to text.

For this blog we will pick up an email spam filtering dataset where the goal is to classify whether an email is spam or not. Let’s look at our data: -

|  |
| --- |
| import numpy as np import pandas as pd df = pd.read\_csv('/content/train-dataset.csv')  # shuffling all our data df = df.sample(frac=1)  # reading only Message\_body and label df = df[['Message\_body','Label']] df |



### Lower casing

* 1. It is a text pre-processing technique where all words are lower cased so that words like ‘cat’ and ‘CAT’ are treated the same way. This technique comes in handy while we are using **Bag of words** or **Tf-Idf** for making features out of our natural language data.
  2. This might not be helpful while doing Part of Speech Tagging ( where nouns can be differentiated based on the case of text ) or Sentiment Analysis ( where Capital words generally depict anger ) it is recommended not to use this technique for your text pre-processing.

|  |
| --- |
| df['clean\_msg']= df['clean\_msg'].apply(lambda x: x.lower()) |

### Removal of Punctuations

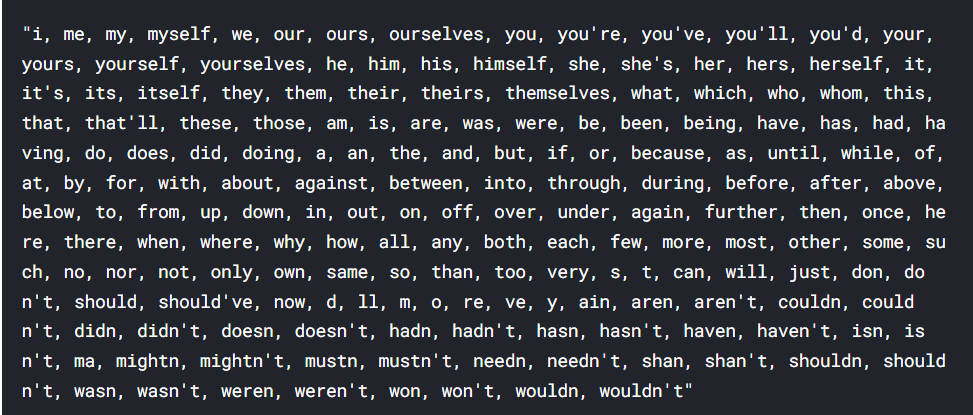
* 1. It is also a technique of text pre-processing where we try to remove unnecessary punctuation symbols because their presence does not make any significance in our text data.  
     For Example, *‘yippee’* and *‘yippee!’* are conveying the feeling of happiness and excitement and this exclamation mark is of no use here.
  2. We will punctuation marks from string.punctuation
  3. But you always add more symbols based on your use case.

|  |
| --- |
| #library that contains punctuation import string # list of all punctuations we have print(string.punctuation)  #defining the function to remove punctuation def remove\_punctuation(text):  punctuationfree="".join([i for i in text if i not in string.punctuation])  return punctuationfree  #storing the punctuation free text for both training and testing data df['clean\_msg']= df['clean\_msg'].apply(lambda x:remove\_punctuation(x)) |

### Removal of Stopwords

* 1. Stopwords are a set of words that do not value a text example ‘a’,’an’,’the’ these are the words which occur very frequently in our text data, but they are of no use. Many libraries have compiled stop words for various languages and we can use them directly and for any specific use case if we feel we can also add a more specific set of stop words to the list.

|  |
| --- |
| from nltk.corpus import stopwords ", ".join(stopwords.words('english')) |

****

* 1. Code for removal of stop words

Before stop words removal we need tokenized text

|  |
| --- |
| #defining function for tokenization import re #whitespace tokenizer from nltk.tokenize import WhitespaceTokenizer def tokenization(text):  tk = WhitespaceTokenizer()  return tk.tokenize(text)  #applying function to the column for making tokens in both Training and Testing data df['tokenised\_clean\_msg']= df['clean\_msg'].apply(lambda x: tokenization(x)) |

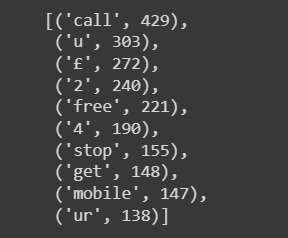
Now stop words removal

|  |
| --- |
| #importing nlp library import nltk nltk.download('stopwords') #Stop words present in the library stopwords = nltk.corpus.stopwords.words('english')  #defining the function to remove stopwords from tokenized text def remove\_stopwords(text):  output= [i for i in text if i not in stopwords]  return output  #applying the function for removal of stopwords df['cleaned\_tokens']= df['tokenised\_clean\_msg'].apply(lambda x:remove\_stopwords(x)) |

### Removal of Frequent words

* 1. As we have removed stop words but in some cases it’s better to remove most frequent words from the data itself that are useless. The most frequent words in our corpus are :-

|  |
| --- |
| from collections import Counter cnt = Counter() for text in df["cleaned\_tokens"].values:  for word in text:  cnt[word] += 1   cnt.most\_common(10) |



|  |
| --- |
| from collections import Counter cnt = Counter() for text in df["cleaned\_tokens"].values:  for word in text:  cnt[word] += 1  FREQWORDS = set([w for (w, wc) in cnt.most\_common(10)])  def remove\_freqwords(text):  """custom function to remove the frequent words"""  return " ".join([word for word in text if word not in FREQWORDS]) |

### Stemming

* 1. It is a text standardisation technique where a word is reduced to its stem/base word. Example: “jabbing” → “jab” and “kicking” → “kick”. The main aim for stemming is that we can reduce the vocab size before inputting it into any machine learning model.

|  |
| --- |
| #importing the Stemming function from nltk library from nltk.stem.porter import PorterStemmer #defining the object for stemming porter\_stemmer = PorterStemmer()  #defining a function for stemming def stemming(text):  stem\_text = [porter\_stemmer.stem(word) for word in text]  return stem\_text  # applying function for stemming df['cleaned\_tokens']=df['cleaned\_tokens'].apply(lambda x: stemming(x)) |

* 1. The Disadvantage of stemming is sometimes after the stemming word loses its meaning. Example: – “copying” → “copi” and there is no word “copi” in English vocab.
  2. Also this porter stemmer is for the English language. If we are working with other languages, we can use **snowball stemmer**.

### Lemmatization

* 1. Lemmatization is very similar to stemming with the only difference being the word here will get reduced to a word that has a particular meaning in its language. Due to this, lemmatization is generally slower than stemming.
  2. Let us use the WordNetLemmatizer in nltk to lemmatize our sentences.

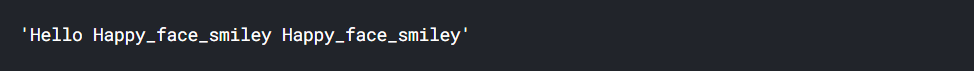
.

|  |
| --- |
| from nltk.stem import WordNetLemmatizer  lemmatizer = WordNetLemmatizer() def lemmatize\_words(text):  return " ".join([lemmatizer.lemmatize(word) for word in text.split()]) |

### Conversion of Emoticons

* 1. We know that on social media there is continuously increasing use of emoticons so it’s better to convert these emoticons back to some natural text so that we can get some useful content out of them.
  2. This method can be useful for some use cases.
  3. For implementation refer to the notebook at the end.

|  |
| --- |
| def convert\_emoticons(text):  for emot in EMOTICONS:  text = re.sub(u'('+emot+')', "\_".join(EMOTICONS[emot].replace(",","").split()), text)  return text  text = "Hello :-) :-)" convert\_emoticons(text) |



### Removal of URL’S

* 1. Next pre-processing step is to remove any URLs present in the data. For example, if we are doing a news data analysis, then there is a very good chance that the news article will have some URL in it. Probably we might need to remove them for our further analysis. We can use the below code snippet to do that.
  2. For example, we will just remove https links

|  |
| --- |
| def remove\_urls(text):  url\_pattern = re.compile(r'https?://\S+|www\.\S+')  return url\_pattern.sub(r'', text) |

### Remove HTML tags

* 1. While scrapping data from different websites, there are very high chances that we might get html tags with that and it’s useful to remove those html tags for any further processing. We’ll use regular expressions to remove html tags

|  |
| --- |
| def remove\_html(text):  html\_pattern = re.compile('<.\*?>')  return html\_pattern.sub(r'', text)  text = """<div> <h1> Data</h1> <p> News articles</p> """  print(remove\_html(text)) |

Output :-

|  |
| --- |
| Data News articles |

### Spelling Correction

* 1. Typos are very common when it comes to either social media or when users on different platforms unintentionally type wrong spellings on platforms and it is extremely useful to correct wrong spellings so that you can make better analysis of textual data.
  2. If we are interested in writing a spell corrector of our own, we can probably start with the famous code from [Peter Norvig](https://norvig.com/spell-correct.html).

|  |
| --- |
| from spellchecker import SpellChecker  spell = SpellChecker() def correct\_spellings(text):  corrected\_text = []  misspelled\_words = spell.unknown(text.split())  for word in text.split():  if word in misspelled\_words:  corrected\_text.append(spell.correction(word))  else:  corrected\_text.append(word)  return " ".join(corrected\_text) |

### Removal of Rare words

* 1. This approach is very specific and, in some cases, can give you good scores based on your metrics.

|  |
| --- |
| from collections import Counter cnt = Counter() for text in df["cleaned\_tokens"].values:  for word in text:  cnt[word] += 1  n\_rare\_words = 10  Rare\_words = set([w for (w, wc) in cnt.most\_common()[:-n\_rare\_words-1:-1]]) def remove\_rarewords(text):  """custom function to remove the rare words"""  return " ".join([word for word in str(text).split() if word not in Rare\_words]) |

So these are some of the text pre-processing techniques that you can use whenever you are dealing with textual data and based on your data text pre-processing techniques can change and you can make your own text pre-processing techniques.

### Why are they useful?

To understand the impact of these techniques we will pick up a problem and will compare the results with and without text pre-processing.

The problem we will see today is of spam filtering in emails and we will use Bag of words model

That we described in detail over [here](https://www.naukri.com/learning/articles/spam-filtering-using-bag-of-words/)  but briefly **Bag of words** is just a Vectorisation algorithm where a text document is converted to a bag of words vector by counting how many times a word appears in a text document.

For this problem I have used few of the text pre-processing techniques from above and we got a good improvement in our results

Results time: -

1. When we did not use any text pre-processing, we got an accuracy of nearly ~88%.
   1. You can check a complete notebook below
2. When we used text pre-processing we got an accuracy of nearly ~92% so that means an increase of nearly 4.5% which is fair enough for this dataset.

Hope this blog helps you understand what is text pre-processing and how you can understand your data and make your own text pre-processing techniques.

I have attached both the notebooks below with and without text pre-processing for your convenience.