Basic Text Pre-processing of text data

- Stemming
- Stopwords removal
- Lemmatization
- Lower casing
- Punctuation removal
- Frequent words removal
- Rare words removal
- Spelling correction
- Tokenization

Advance Text Processing

- Bag of Words
- N-grams
- Term Frequency
- Inverse Document Frequency
- Term Frequency-Inverse Document Frequency (TF-IDF)
- Word2Vec
- Avg Word2Vec
- TF-IDF Weighted Word2Vec
- Sentiment Analysis

Basic Text Pre-processing of text data

1. Stemming

Stemming refers to the removal of suffices, like "ing", "ly", "s", etc. by a simple rule-based approach. For this purpose, we will use PorterStemmer from the NLTK library.

```
from nltk.stem import PorterStemmer
st = PorterStemmer()
train['tweet'][:5].apply(lambda x: " ".join([st.stem(word) for word in x.split()]))
```

```
0        father dysfunct selfish drag kid dysfunct run
1    thank lyft credit cant use caus dont offer whe...
2                                     bihday majesti
3                             model take urð ðððð ðð
4                             factsguid societi motiv
Name: tweet, dtype: object
```

In the above output, dysfunctional has been transformed into dysfunct, among other changes.

2.Lemmatization

Lemmatization is a more effective option than stemming because it converts the word into its root word, rather than just stripping the suffices.

- It makes use of the vocabulary and does a morphological analysis to obtain the root word.

```
from textblob import Word
train['tweet'] = train['tweet'].apply(lambda x: " ".join([Word(word).lemmatize() fo
r word in x.split()]))
train['tweet'].head()
```

3. Removal of Stop Words

Stop words (or commonly occurring words) should be removed from the text data. For this purpose, we can either create a list of stopwords ourselves or we can use predefined libraries.

```
from nltk.corpus import stopwords
stop = stopwords.words('english')
train['tweet'] = train['tweet'].apply(lambda x: " ".join(x for x in x.split() if x
not in stop))
train['tweet'].head()
```

```
0    user father dysfunctional selfish drags kids d...
1    user user thanks lyft credit cant use cause do...
2                                      bihday majesty
3                model love u take u time urð ðððð ðð
4                        factsguide society motivation
Name: tweet, dtype: object
```

4.Tokenization

Tokenization refers to dividing the text into a sequence of words or sentences. In our example, we have used the textblob library to first transform our tweets into a blob and then converted them into a series of words.

Advance Text Processing -

Need to Convert Text into Vectors

We can understood that sentence in a fraction of a second. But machines simply cannot process text data in raw form. They need us to break down the text into a numerical format which is easily readable by the machine.

Word Embedding is one such technique where we can represent the text using vectors. The more popular forms of word embeddings are:

1. BoW, which stands for Bag of Words
2. TF-IDF, which stands for Term Frequency-Inverse Document Frequency

1. Bag of Words(BOG)

It's one of the simplest methods in text processing that converts text into vectors. It is a representation of word occurrence in data.
It doesn't care about the information in data. It only cares about frequencies of words in data that's why it's called **BAG.**

Eg -

- Review 1: This movie is very scary and long
- Review 2: This movie is not scary and is slow
- Review 3: This movie is spooky and good

We will first build a vocabulary from all the unique words in the above three reviews. The vocabulary consists of these 11 words: 'This', 'movie', 'is', 'very', 'scary', 'and', 'long', 'not', 'slow', 'spooky', 'good'.

We can now take each of these words and mark their occurrence in the three movie reviews above with 1s and 0s. This will give us 3 vectors for 3 reviews-

	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good	Length of the review(in words)
Review 1	1	1	1	1	1	1	1	0	0	0	0	7
Review 2	1	1	2	0	0	1	1	0	1	0	0	8
Review 3	1	1	1	0	0	0	1	0	0	1	1	6

Vector of Review 1: [1 1 1 1 1 1 1 0 0 0 0]
Vector of Review 2: [1 1 2 0 0 1 1 0 1 0 0]
Vector of Review 3: [1 1 1 0 0 0 1 0 0 1 1]

And that's the core idea behind a Bag of Words (BoW) model.

Drawbacks of using a Bag-of-Words (BoW) Model

In the above example, we can have vectors of length 11. However, we start facing issues when we come across new sentences:

1. If the new sentences contain new words, then our vocabulary size would increase and thereby, the length of the vectors would increase too.
2. The vectors would also contain many 0s, thereby resulting in a sparse matrix (which is what we would like to avoid)
3. We are retaining no information on the grammar of the sentences nor on the ordering of the words in the text.

2. GRAMS

GRAMS are also used to convert text to vectors. First, it converts the word into groups before the process. It can be 2, 3, or n groups. In this technique, words are called Gram.
Let's see what it means-
Sentence: I am a writer at medium and write articles on Machine learning.
Uni-gram — It takes only one word at a time to convert into vectors.

I	*am*	a	writer	at	Medium	and	write	articles	on	machine	learning

Bi-gram —— It takes two words at a time to convert into vectors.

I am	*am a*	a writer

Writer at	At medium	Medium and	And write	write articles	Article on	On machine	Machine learning	learning

Tri-gram — It takes three words at a time to convert into vectors. N-gram — It takes N-words at a time to convert into vectors. **Grams saved sentence information too.**

3. Term Frequency(TF)

- It used to convert text into vectors.
- It is a measure of **how frequently** a term appears in a document
  TF = number of times the word appears in the data/total number of words in the data
  Eg -
  Review 2: This movie is not scary and is slow

Here,

- Vocabulary- 'This', 'movie', 'is', 'very', 'scary', 'and', 'long', 'not', 'slow', 'spooky', 'good'
- Number of words in Review 2 = 8
- TF for the word 'this' = (number of times 'this' appears in review 2)/(number of terms in review 2) = 1/8

Similarly,

TF('movie') = 1/8 , TF('is') = 2/8 = 1/4 , TF('very') = 0/8 = 0 , TF('scary') = 1/8 , TF('and') = 1/8 , TF('long') = 0/8 = 0 , TF('not') = 1/8 , TF('slow') = 1/8 , TF('spooky') = 0/8 = 0 , TF('good') = 0/8 = 0 ,

We can calculate the term frequencies for all the terms and all the reviews in this manner-

Term	Review 1	Review 2	Review 3	TF (Review 1)	TF (Review 2)	TF (Review 3)
This	1	1	1	1/7	1/8	1/6
movie	1	1	1	1/7	1/8	1/6
is	1	2	1	1/7	1/4	1/6
very	1	0	0	1/7	0	0
scary	1	1	0	1/7	1/8	0
and	1	1	1	1/7	1/8	1/6
long	1	0	0	1/7	0	0
not	0	1	0	0	1/8	0
slow	0	1	0	0	1/8	0
spooky	0	0	1	0	0	1/6
good	0	0	1	0	0	1/6

4. Inverse Data Frequency (IDF)

IDF is a measure of **how important** a term is. We need the IDF value because computing just the TF alone is not sufficient to understand the importance of words-

$$idf_t = \log \frac{number\ of\ documents}{number\ of\ documents\ with\ term\ 't'}$$

Eg-We can calculate the IDF values for the all the words in Review 2:

IDF('this') = log(number of documents/number of documents containing the word 'this') = log(3/3) = log(1) = 0

Similarly,

IDF('movie',) = log(3/3) = 0 , IDF('is') = log(3/3) = 0 , IDF('not') = log(3/1) = log(3) = 0.48 , IDF('scary') = log(3/2) = 0.18 , IDF('and') = log(3/3) = 0 , IDF('slow') = log(3/1) = 0.48 .

We can calculate the IDF values for each word like this. Thus, the IDF values for the entire vocabulary would be-

Term	Review 1	Review 2	Review 3	IDF
This	1	1	1	0.00
movie	1	1	1	0.00
is	1	2	1	0.00
very	1	0	0	0.48
scary	1	1	0	0.18
and	1	1	1	0.00
long	1	0	0	0.48
not	0	1	0	0.48
slow	0	1	0	0.48
spooky	0	0	1	0.48
good	0	0	1	0.48

Hence, we see that words like "is", "this", "and", etc., are reduced to 0 and have little importance, while words like "scary", "long", "good", etc. are words with **more importance and thus have a higher value.**

5. Term Frequency–Inverse Document Frequency (TF-IDF)

TF-IDF is a numerical statistic that tells how important a word is to a document in a collection or corpus.
As we discussed above -

$$TF = \frac{Number - of - times - the - word - appears - in - the - data}{Total - number - of - words - in - the - data}$$

$$IDF = log(\frac{Number - of - Docs}{Number - Docs - the - term - appears - in})$$

$$TF - ID = TF * IDF$$

We can now compute the TF-IDF score for each word in the corpus.
Words with a **higher score are more important**, and those with a lower score are less important

Eg-

We can now calculate the TF-IDF score for every word in Review 2:

TF-IDF('this', Review 2) = TF('this', Review 2) * IDF('this') = 1/8.0 = 0
Similarly,

Term	Review 1	Review 2	Review 3	IDF	TF-IDF (Review 1)	TF-IDF (Review 2)	TF-IDF (Review 3)
This	1	1	1	0.00	0.000	0.000	0.000
movie	1	1	1	0.00	0.000	0.000	0.000
is	1	2	1	0.00	0.000	0.000	0.000
very	1	0	0	0.48	0.068	0.000	0.000
scary	1	1	0	0.18	0.025	0.022	0.000
and	1	1	1	0.00	0.000	0.000	0.000
long	1	0	0	0.48	0.068	0.000	0.000
not	0	1	0	0.48	0.000	0.060	0.000
slow	0	1	0	0.48	0.000	0.060	0.000
spooky	0	0	1	0.48	0.000	0.000	0.080
good	0	0	1	0.48	0.000	0.000	0.080

We have now obtained the TF-IDF scores for our vocabulary. TF-IDF also gives larger values for less frequent words and is high when both IDF and TF values are high i.e the word is rare in all the documents combined but frequent in a single document.

6. Word 2 Vec -

Word2vec is a two-layer neural net that processes text by "vectorizing" words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus.

- It convert a word into vector not sentence.
- It preserve semantic meaning
- It learns relationship automatically from raw-text.

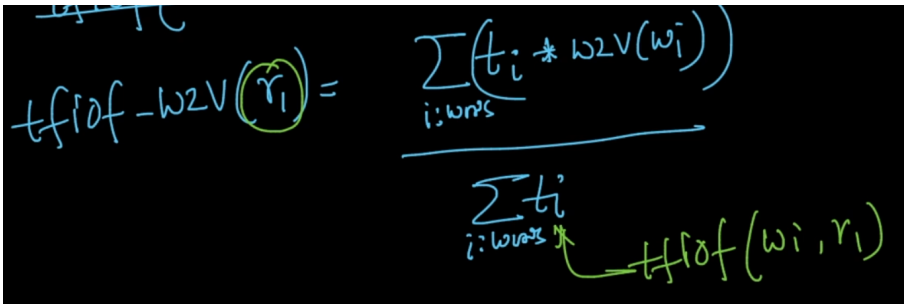
7. Avg Word 2 Vec -

- Avg-Word 2 Vec is used to convert a sentence into vector.

- As sentence is a group of words, here we apply word2vec in every word and take average, then we get Avg Word2Vec of sentence.

8. TF-IDF-Weighted-Word2Vec -

- It is used to convert sentence into vector.
- TF-IDF-Weighted-Word2Vec = [TF-IDF(w) * W2V(w)]/ sum (TF-IDF(wn))



9. Sentiment Analysis -

If you recall, our problem was to detect the sentiment of the tweet. So, before applying any ML/DL models (which can have a separate feature detecting the sentiment using the textblob library), let's check the sentiment of the first few tweets.

```
train['tweet'][:5].apply(lambda x: TextBlob(x).sentiment)
0    (-0.3, 0.5354166666666667)
1                    (0.2, 0.2)
2                    (0.0, 0.0)
3                    (0.0, 0.0)
4                    (0.0, 0.0)
Name: tweet, dtype: object
```

- Above, you can see that it returns a tuple representing polarity and subjectivity of each tweet.
- Here, we only extract polarity as it indicates the sentiment as value **nearer to 1 means a positive sentiment** and values **nearer to -1 means a negative sentiment.**
- This can also work as a feature for building a machine learning model.

```
train['sentiment'] = train['tweet'].apply(lambda x: TextBlob(x).sentiment[0] )
train[['tweet','sentiment']].head()
```

	tweet	sentiment
0	father dysfunctional selfish drag kid dysfunct...	-0.3
1	thanks lyft credit cant use cause dont offer w...	0.2
2	bihday majesty	0.0
3	model take urð ðððð ðð	0.0
4	factsguide society motivation	0.0

Refer -

1. Word Embeddings- https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/ (https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/)
2. Pre-processing Code- https://www.analyticsvidhya.com/blog/2018/02/the-different-methods-deal-text-data-predictive-python/ (https://www.analyticsvidhya.com/blog/2018/02/the-different-methods-deal-text-data-predictive-python/)
3. Full NLP- https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/ (https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/)