Bag of Words (BoW) Model.

The Bag of Words (BoW) model is the simplest form of text representation in numbers. Like the term itself, we can represent a sentence as a bag of words vector or we can simply a string of numbers.

Let's recall the three types of movie reviews :

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

In [164]:
```
import nltk
from nltk.stem import PorterStemmer # used for stemming purpose
from nltk.corpus import stopwords
import nltk
from nltk import sent_tokenize
from nltk import word_tokenize
nltk.download('punkt')
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\panka\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

In [163]:
```
###
Like above we can perform Bow in the given text, result may differ due to some dependiences , but the core principle is intuitative
```

In [137]:
```
para = '''This movie is very scary and long.
          This movie is not scary and is slow.
          This movie is spooky and good.'''
```

In [138]:
```
nltk.download('punkt')
sentences1 = nltk.sent_tokenize(para)
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\panka\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

In [168]:
```
import re
corpus1 = []
for i in range(len(sentences1)):
    review = re.sub('^[a-zA-Z]', ' ',sentences1[i])
    review = review.lower()
    review = review.split()
    review = [lemmatizer.lemmatize(word) for word in review if not word in set(stopwords.words('english'))]
    review = ' '.join(review)
    corpus1.append(review)
```

In [169]:
```
corpus1
```

Out[169]:
```
['movie scary long.', 'movie scary slow.', 'movie spooky good.']
```

In [141]:
```
## Bag of words
from sklearn.feature_extraction.text import CountVectorizer
cv1 = CountVectorizer(ngram_range=(1,1))  # binary=False # ngram_range=(3,3) - try grams
```

In [142]:
```
X1=cv1.fit_transform(corpus1)
```

In [143]:
```
cv1.vocabulary_
```

Out[143]:
```
{'his': 2,
 'movie': 5,
 'is': 3,
 'very': 10,
 'scary': 7,
 'and': 0,
 'long': 4,
 'not': 6,
 'slow': 8,
 'spooky': 9,
 'good': 1}
```

In [144]:
```
X1.toarray()
```

Out[144]:
```
array([[1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1],
       [1, 0, 1, 2, 0, 1, 1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0]], dtype=int64)
```

In [145]:
```
# for sentence 1
X1[0].toarray()

# This movie is very scary and long

#['good': 0,  'long': 1, 'movie': 2, 'scary': 3, 'slow': 4, 'spooky': 5]
```

Out[145]:
```
array([[1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1]], dtype=int64)
```

In [107]:
```
# for sentence 2
X1[1].toarray()

# This movie is not scary and is slow.

#['good': 0,  'long': 1, 'movie': 2, 'scary': 3, 'slow': 4, 'spooky': 5]
```

Out[107]:
```
array([[1, 0, 1, 2, 0, 1, 1, 1, 1, 0, 0]], dtype=int64)
```

In [108]:
```
# for sentence 3
X1[2].toarray()

# This movie is spooky and good.

#['good': 0,  'long': 1, 'movie': 2, 'scary': 3, 'slow': 4, 'spooky': 5]
```

Out[108]:
```
array([[1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0]], dtype=int64)
```

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.Additionally, 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.\

In [ ]:
```

```

TF-IDF

“Term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.”

Term Frequency (TF)

Let’s first understand Term Frequent (TF). It is a measure of how frequently a term, t, appears in a document

$$tf_{t,d} = \frac{n_{t,d}}{Number\ of\ terms\ in\ the\ document}$$

```
Here, in the numerator, n is the number of times the term “t” appears in the document “d”. Thus, each document and term would have its own TF value.
```

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:

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

.

After that we have to 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:

```
<img src='tf_idf.jpg'>
```

In [151]:
```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
```

In [155]:
```
#apply Stopwords lemmatize

import re
corpus = []
for i in range(len(sentences1)):
    review = re.sub('^[a-zA-Z]', ' ',sentences1[i])
    review = review.lower()
    review = review.split()
    review = [lemmatizer.lemmatize(word) for word in review if not word in set(stopwords.words('english'))]
    review = ' '.join(review)
    corpus.append(review)
```

In [156]:
```
X=cv.fit_transform(corpus)
```

In [157]:
```
cv.vocabulary_
```

Out[157]:
```
{'movie scary long': 0, 'movie scary slow': 1, 'movie spooky good': 2}
```

In [158]:
```
X[0].toarray()
```

Out[158]:
```
array([[1., 0., 0.]])
```

In [159]:
```
# TF-IDF
from sklearn.feature_extraction.text import TfidfVectorizer
cv = TfidfVectorizer(ngram_range=(3,3)) #,max_features=10
X1 = cv.fit_transform(corpus1)
```

In [160]:
```
corpus1[0]
```

Out[160]:
```
'his movie is very scary and long.'
```

In [162]:
```
X1.toarray()
```

Out[162]:
```
array([[0.        , 0.28321692, 0.        , 0.        , 0.47952794,
        0.        , 0.        , 0.47952794, 0.        , 0.        ,
        0.47952794, 0.        , 0.47952794],
       [0.43238509, 0.2553736 , 0.43238509, 0.        , 0.        ,
        0.43238509, 0.        , 0.        , 0.43238509, 0.43238509,
        0.        , 0.        , 0.        ],
       [0.        , 0.32274454, 0.        , 0.54645401, 0.        ,
        0.        , 0.54645401, 0.        , 0.        , 0.        ,
        0.        , 0.54645401, 0.        ]])
```

In [161]:
```
X1[0].toarray()
```

Out[161]:
```
array([[0.        , 0.28321692, 0.        , 0.        , 0.47952794,
        0.        , 0.        , 0.47952794, 0.        , 0.        ,
        0.47952794, 0.        , 0.47952794]])
```

In [165]:
```
X1[1].toarray()
```

Out[165]:
```
array([[0.43238509, 0.2553736 , 0.43238509, 0.        , 0.        ,
        0.43238509, 0.        , 0.        , 0.43238509, 0.43238509,
        0.        , 0.        , 0.        ]])
```

In [166]:
```
X1[2].toarray()
```

Out[166]:
```
array([[0.        , 0.32274454, 0.        , 0.54645401, 0.        ,
        0.        , 0.54645401, 0.        , 0.        , 0.        ,
        0.        , 0.54645401, 0.        ]])
```

Advantages and Disadvantages of TF-IDF

Advantages:

- Easy to compute
- You have some basic metric to extract the most descriptive terms in a document
- You can easily compute the similarity between 2 documents using it

Disadvantages:

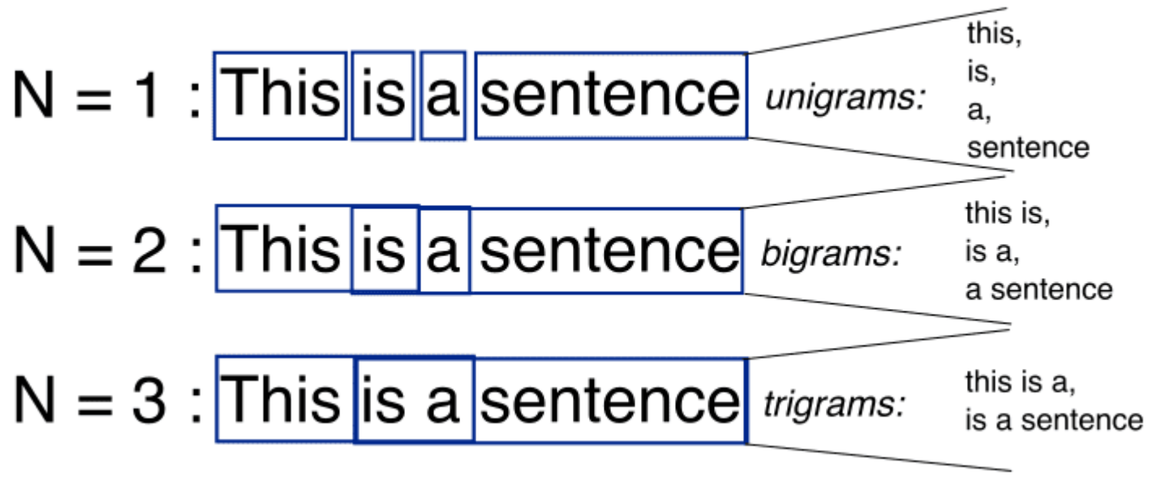
- TF-IDF is based on the bag-of-words (BoW) model, therefore it does not capture position in text, semantics, co-occurrences in different documents, etc.
- For this reason, TF-IDF is only useful as a lexical level feature
- Cannot capture semantics (e.g. as compared to topic models, word embeddings)

In [ ]:
```
```

N-grams Model:

A more sophisticated approach is to create a vocabulary of grouped words. This changes both the scope of the vocabulary and allows the bag-of-words to capture a little bit more meaning from the document.

In this approach, each word or token is called a “gram”. Creating a vocabulary of two-word pairs is, in turn, called a bigram model. Again, only the bigrams that appear in the corpus are modeled, not all possible bigrams.



Unigrams or 1-grams

In [182]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
cv = TfidfVectorizer() #,max_features=10
X = cv.fit_transform(corpus1)
```

In [183]:
```
cv.vocabulary_
```

Out[183]:
```
{'movie': 2, 'scary': 3, 'long': 1, 'slow': 4, 'spooky': 5, 'good': 0}
```

Bigrams or 2-grams

In [184]:
```
cv2 = TfidfVectorizer(ngram_range=(2,2)) #,max_features=7
X2 = cv2.fit_transform(corpus1)
```

In [185]:
```
cv2.vocabulary_
```

Out[185]:
```
{'movie scary': 0,
 'scary long': 2,
 'scary slow': 3,
 'movie spooky': 1,
 'spooky good': 4}
```

Trigrams or 3-grams

In [186]:
```
cv3 = TfidfVectorizer(ngram_range=(3,3)) #,max_features=10
X3 = cv3.fit_transform(corpus1)
```

In [187]:
```
cv3.vocabulary_
```

Out[187]:
```
{'movie scary long': 0, 'movie scary slow': 1, 'movie spooky good': 2}
```

Applications of the N-gram Model in NLP

1.For N-1 words, the N-gram modeling predicts most occurred words that can follow the sequences. The model is the probabilistic language model which is trained on the collection of the text. This model is useful in applications i.e. speech recognition, and machine translations. A simple model has some limitations that can be improved by smoothing, interpolations, and back off. So, the N-gram language model is about finding probability distributions over the sequences of the word.

2.Using these n-grams and the probabilities of the occurrences of certain words in certain sequences could improve the predictions of auto completion systems. Similarly, we use can NLP and n-grams to train voice-based personal assistant bots. For example, using a 3-gram or trigram training model, a bot will be able to understand the difference between sentences such as “what’s the temperature?” and “set the temperature.”

Limitations of N-gram Model in NLP

The N-gram language model has also some limitations. There is a problem with the out of vocabulary words. These words are during the testing but not in the training. One solution is to use the fixed vocabulary and then convert out vocabulary words in the training to pseudowords. When implemented in the sentiment analysis, the bi-gram model outperformed the uni-gram model but the number of the features is then doubled. So, the scaling of the N-gram model to the larger data sets or moving to the higher-order needs better feature selection approaches. The N-gram model captures the long-distance context poorly. It has been shown after every 6-grams, the gain of performance is limited

In [ ]:

```
By Pankaj Kuamr Barman
MSc. Computer Science and Machine Intelligence
```