In [1]:
```
# Tokenization using NLTK
from nltk import word_tokenize, sent_tokenize
sent = "For writers, a random sentence can help them get their creative juices
print(word_tokenize(sent))
print(sent_tokenize(sent))
```

```
['For', 'writers', ',', 'a', 'random', 'sentence', 'can', 'help', 'them', 'ge
t', 'their', 'creative', 'juices', 'flowing', '.', 'Since', 'the', 'topic',
'of', 'the', 'sentence', 'is', 'completely', 'unknown', ',', 'it', 'forces',
'the', 'writer', 'to', 'be', 'creative', 'when', 'the', 'sentence', 'appear
s', '.', 'There', 'are', 'a', 'number', 'of', 'different', 'ways', 'a', 'writ
er', 'can', 'use', 'the', 'random', 'sentence', 'for', 'creativity', '.']
['For writers, a random sentence can help them get their creative juices flow
ing.', 'Since the topic of the sentence is completely unknown, it forces the
writer to be creative when the sentence appears.', 'There are a number of dif
ferent ways a writer can use the random sentence for creativity.']
```

In [2]:
```
from nltk.stem import PorterStemmer

# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("play"))
print(porter.stem("playing"))
print(porter.stem("plays"))
print(porter.stem("played"))
```

```
play
play
play
play
```

In [3]:
```
from nltk.stem import PorterStemmer
# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("Communication"))
```

```
commun
```

In [4]:
```
import nltk
from nltk.stem.snowball import SnowballStemmer

#the stemmer requires a language parameter
snow_stemmer = SnowballStemmer(language='english')

#list of tokenized words
words = ['cared','university','fairly','easily','singing',
    'sings','sung','singer','sportingly']

#stem's of each word
stem_words = []
for w in words:
    x = snow_stemmer.stem(w)
    stem_words.append(x)

#print stemming results
for e1,e2 in zip(words,stem_words):
    print(e1+' ----> '+e2)
```

```
cared ----> care
university ----> univers
fairly ----> fair
easily ----> easili
singing ----> sing
sings ----> sing
sung ----> sung
singer ----> singer
sportingly ----> sport
```

In [5]:
```
from nltk.stem import WordNetLemmatizer
# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("plays", 'v'))
print(lemmatizer.lemmatize("played", 'v'))
print(lemmatizer.lemmatize("play", 'v'))
print(lemmatizer.lemmatize("playing", 'v'))
```

```
play
play
play
play
```

In [6]:
```
from nltk.stem import WordNetLemmatizer

# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("Communication", 'v'))
```

```
Communication
```

3 basic approaches in Bag of Words which are better than Word Embeddings

[3 basic approaches in Bag of Words which are better than Word Embeddings (https://towardsdatascience.com/3-basic-approaches-in-bag-of-words-which-are-better-than-word-embeddings-c2cbc7398016)](https://towardsdatascience.com/3-basic-approaches-in-bag-of-words-which-are-better-than-word-embeddings-c2cbc7398016)

3 basic approaches in Bag of Word which are better than Word Embedding

Nowadays, every one is talking about Word (or Character, Sentence, Document) Embeddings. Is Bag of Words still worth using? Should we apply embedding in any scenario? After reading this article, you will know:

- Why people say that Word Embedding is the silver bullet?
- When does Bag of Words win over Word Embeddings?
- 3 basic approaches in Bag of Words
- How can we build Bag of Words in a few line?

Why somebody say that Word Embeddings are the silver bullet?

In the-state-of-art of the NLP field, Embedding is the success way to resolve text related problem and outperform Bag of Words (BoW). Indeed, BoW introduced limitations large feature dimension, sparse representation etc. For word embedding, you may check out my previous post.

Should we still use BoW? We may better use BoW in some scenarios.

When does Bag of Words win over Word Embeddings?

You may still consider to use BoW rather than Word Embedding in the following situations:

- Building an baseline model. By using scikit-learn, there is just a few lines of code to build model. Later on, can using Deep Learning to bit it.
- If your dataset is small and context is domain specific, BoW may work better than Word Embedding. Context is very domain specific which means that you cannot find corresponding Vector from pre-trained word embedding models (GloVe, fastText etc).

How can we build Bag of Words in a few line?

There is 3 simple ways to build BoW model by using traditional powerful ML libraries.

In [1]:
```
import collections
import pandas as pd
import numpy as np

from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_score, KFold
```

In [2]:
```
from sklearn.datasets import fetch_20newsgroups
train_raw_df = fetch_20newsgroups(subset='train')
```

In [3]:
```
x_train = train_raw_df.data
y_train = train_raw_df.target
```

Count Occurrence

Photo: https://pixabay.com/en/home-money-euro-calculator-finance-366927/ (https://pixabay.com/en/home-money-euro-calculator-finance-366927/)

Counting word occurrence. The reason behind of using this approach is that keyword or important signal will occur again and again. So if the number of occurrence represent the importance of word. More frequency means more importance.

In [4]:
```
doc = "In the-state-of-art of the NLP field, Embedding is the \
success way to resolve text related problem and outperform \
Bag of Words ( BoW ). Indeed, BoW introduced limitations \
large feature dimension, sparse representation etc."

count_vec = CountVectorizer()
count_occurs = count_vec.fit_transform([doc])
count_occur_df = pd.DataFrame((count, word) for word, count in zip(count_occur
count_occur_df.columns = ['Word', 'Count']
count_occur_df.sort_values('Count', ascending=False, inplace=True)
count_occur_df.head()
```

Out[4]:

	Word	Count
16	of	3
26	the	3
3	bow	2
0	and	1
28	way	1

Normalized Count Occurrence

If you think that high frequency may dominate the result and causing model bias. Normalization can be apply to pipeline easily.

In [5]:
```
doc = "In the-state-of-art of the NLP field, Embedding is the \
success way to resolve text related problem and outperform \
Bag of Words ( BoW ). Indeed, BoW introduced limitations \
large feature dimension, sparse representation etc."

norm_count_vec = TfidfVectorizer(use_idf=False, norm='l2')
norm_count_occurs = norm_count_vec.fit_transform([doc])
norm_count_occur_df = pd.DataFrame((count, word) for word, count in zip(
    norm_count_occurs.toarray().tolist()[0], norm_count_vec.get_feature_names(
norm_count_occur_df.columns = ['Word', 'Count']
norm_count_occur_df.sort_values('Count', ascending=False, inplace=True)
norm_count_occur_df.head()
```

Out[5]:

	Word	Count
16	of	0.428571
26	the	0.428571
3	bow	0.285714
0	and	0.142857
28	way	0.142857

TF-IDF

TF-IDF take another approach which is believe that high frequency may not able to provide much information gain. In another word, rare words contribute more weights to the model.

Word importance will be increased if the number of occurrence within same document (i.e. training record). On the other hand, it will be decreased if it occurs in corpus (i.e. other training records).

In [6]:

```
doc = "In the-state-of-art of the NLP field, Embedding is the \
success way to resolve text related problem and outperform \
Bag of Words ( BoW ). Indeed, BoW introduced limitations \
large feature dimension, sparse representation etc."

tfidf_vec = TfidfVectorizer()
tfidf_count_occurs = tfidf_vec.fit_transform([doc])
tfidf_count_occur_df = pd.DataFrame((count, word) for word, count in zip(
    tfidf_count_occurs.toarray().tolist()[0], tfidf_vec.get_feature_names()))
tfidf_count_occur_df.columns = ['Word', 'Count']
tfidf_count_occur_df.sort_values('Count', ascending=False, inplace=True)
tfidf_count_occur_df.head()
```

Out[6]:

	Word	Count
16	of	0.428571
26	the	0.428571
3	bow	0.285714
0	and	0.142857
28	way	0.142857

Preprocessing

In [7]:
```
stop_words = ['a', 'an', 'the']

# Basic cleansing
def cleansing(text):
    # Tokenize
    tokens = text.split(' ')
    # Lower case
    tokens = [w.lower() for w in tokens]
    # Remove stop words
    tokens = [w for w in tokens if w not in stop_words]
    return ' '.join(tokens)

# All-in-one preproce
def preprocess_x(x):
    processed_x = [cleansing(text) for text in x]

    return processed_x

def build_model(mode):
    # Intent to use default paramaters for show case
    vect = None
    if mode == 'count':
        vect = CountVectorizer()
    elif mode == 'tf':
        vect = TfidfVectorizer(use_idf=False, norm='l2')
    elif mode == 'tfidf':
        vect = TfidfVectorizer()
    else:
        raise ValueError('Mode should be either count or tfidf')

    return Pipeline([
        ('vect', vect),
        ('clf' , LogisticRegression(solver='newton-cg',n_jobs=-1))
    ])

def pipeline(x, y, mode):
    processed_x = preprocess_x(x)

    model_pipeline = build_model(mode)
    cv = KFold(n_splits=5, shuffle=True)

    scores = cross_val_score(model_pipeline, processed_x, y, cv=cv, scoring='a
    print("Accuracy: %0.4f (+/- %0.4f)" % (scores.mean(), scores.std() * 2))

    return model_pipeline
```

Let check number of vocabulary we need to handle

In [ ]:
```
x = preprocess_x(x_train)
y = y_train

model_pipeline = build_model(mode='count')
model_pipeline.fit(x, y)

print('Number of Vocabulary: %d'% (len(model_pipeline.named_steps['vect'].get_
```

Pipeline

In [ ]:
```
print('Using Count Vectorizer------')
model_pipeline = pipeline(x_train, y_train, mode='count')

print('Using TF Vectorizer------')
model_pipeline = pipeline(x_train, y_train, mode='tf')

print('Using TF-IDF Vectorizer------')
model_pipeline = pipeline(x_train, y_train, mode='tfidf')
```

Conclusion

From previous experience, I tried to tackle the problem of classifying product category by giving a short description. For example, given "Fresh Apple" and the expected category is "Fruit". Already able to have 80+ accuracy by using count occurrence approach only.

In this case, since the number of word per training record is just a few words (from 2 words to 10 words). It may not be a good idea to use Word Embedding as there is no much neighbor (words) for training the vectors. On the other hand, scikit-learn provides other parameter to further tune the model input. You may need to take a look on the following features

- ngram_range: Rather than using single word, ngram can be defined as well
- binary: Besides counting occurrence, binary representation can be chosen.
- max_features: Instead of using all words, max number of word can be chosen to reduce the model complexity and size.

Also, some preprocessing steps can be executed within above library rather than handle it by yourself. For example, stop word removal, lower case etc. To have a better flexibility, I will use my own code to finish the preprocessing steps.

[Dataset link (https://www.kaggle.com/CooperUnion/cardataset)](https://www.kaggle.com/CooperUnion/cardataset)

[Reference (https://towardsdatascience.com/a-beginners-guide-to-word-embedding-with-gensim-word2vec-model-5970fa56cc92)](https://towardsdatascience.com/a-beginners-guide-to-word-embedding-with-gensim-word2vec-model-5970fa56cc92)

Introduction to Word2Vec

Word2vec is one of the most popular technique to learn word embeddings using a two-layer neural network. Its input is a text corpus and its output is a set of vectors. Word embedding via word2vec can make natural language computer-readable, then further implementation of mathematical operations on words can be used to detect their similarities. A well-trained set of word vectors will place similar words close to each other in that space. For instance, the words women, men, and human might cluster in one corner, while yellow, red and blue cluster together in another.

There are two main training algorithms for word2vec, one is the continuous bag of words(CBOW), another is called skip-gram. The major difference between these two methods is that CBOW is using context to predict a target word while skip-gram is using a word to predict a target context. Generally, the skip-gram method can have a better performance compared with CBOW method, for it can capture two semantics for a single word. For instance, it will have two vector representations for Apple, one for the company and another for the fruit. For more details about the word2vec algorithm, please check [here (https://arxiv.org/pdf/1301.3781.pdf)](https://arxiv.org/pdf/1301.3781.pdf).

Gensim Python Library Introduction

Gensim is an open source python library for natural language processing and it was developed and is maintained by the Czech natural language processing researcher Radim Řehůřek. Gensim library will enable us to develop word embeddings by training our own word2vec models on a custom corpus either with CBOW of skip-grams algorithms.

In [1]:
```
!pip install --upgrade gensim
```

```
Collecting gensim
  Downloading gensim-4.3.1-cp38-cp38-win_amd64.whl (24.0 MB)
Requirement already satisfied: smart-open>=1.8.1 in d:\program files\anaconda
3\lib\site-packages (from gensim) (6.3.0)
Requirement already satisfied: scipy>=1.7.0 in d:\program files\anaconda3\lib
\site-packages (from gensim) (1.10.1)
Requirement already satisfied: numpy>=1.18.5 in d:\program files\anaconda3\li
b\site-packages (from gensim) (1.23.5)
Installing collected packages: gensim
Successfully installed gensim-4.3.1
```

Download the data

Dataset Description

This vehicle dataset includes features such as make, model, year, engine, and other properties of the car. We will use these features to generate the word embeddings for each make model and then compare the similarities between different make model.

In [2]:
```
!wget https://raw.githubusercontent.com/PICT-NLP/BE-NLP-Elective/main/2-Embedd
```

```
'wget' is not recognized as an internal or external command,
operable program or batch file.
```

Implementation of Word Embedding with Gensim

In [3]:
```
import pandas as pd
```

In [4]:
```
df = pd.read_csv('data.csv')
df.head()
```

Out[4]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	M
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Tu
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Lux
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Lux
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	

Data Preprocessing

Since the purpose of this tutorial is to learn how to generate word embeddings using genism library, we will not do the EDA and feature selection for the word2vec model for the sake of simplicity.

Genism word2vec requires that a format of 'list of lists' for training where every document is contained in a list and every list contains lists of tokens of that document. At first, we need to generate a format of 'list of lists' for training the make model word embedding. To be more specific, each make model is contained in a list and every list contains lists of features of that make model.

To achieve this, we need to do the following things

Create a new column for Make Model

In [5]:
```
df['Maker_Model']= df['Make']+ " " + df['Model']
```

Generate a format of ' list of lists' for each Make Model with the following features: Engine Fuel Type, Transmission Type, Driven_Wheels, Market Category, Vehicle Size, Vehicle Style.

In [6]:
```
df1 = df[['Engine Fuel Type','Transmission Type','Driven_Wheels','Market Categ

df2 = df1.apply(lambda x: ','.join(x.astype(str)), axis=1)

df_clean = pd.DataFrame({'clean': df2})

sent = [row.split(',') for row in df_clean['clean']]
```

Genism word2vec Model Training

We can train the genism word2vec model with our own custom corpus as following:

In [7]:
```
from gensim.models.word2vec import Word2Vec
```

Let's try to understand the hyperparameters of this model.

1. `vector_size` : The number of dimensions of the embeddings and the default is 100.
2. `window` : The maximum distance between a target word and words around the target word. The default window is 5.
3. `min_count` : The minimum count of words to consider when training the model; words with occurrence less than this count will be ignored. The default for min_count is 5.
4. `workers` : The number of partitions during training and the default workers is 3.
5. `sg` : The training algorithm, either CBOW(0) or skip gram(1). The default training algorithm is CBOW.

After training the word2vec model, we can obtain the word embedding directly from the training model as following.

In [8]:
```
model = Word2Vec(sent, min_count=1,vector_size= 50,workers=3, window =3, sg =
```

Save the model

In [9]:
```
model.save("word2vec.model")
```

Load the model

In [10]:
```
model = Word2Vec.load("word2vec.model")
```

After training the word2vec model, we can obtain the word embedding directly from the training model as following.

In [11]:
```
model.wv['Toyota Camry']
```

Out[11]:
```
array([ 0.00060954,  0.10690276,  0.05986022, -0.11626566, -0.05881133,
       -0.18526934,  0.00709567,  0.27551118, -0.10320179, -0.0604336 ,
        0.01600162, -0.01016074,  0.13940156, -0.02224435, -0.08011921,
        0.18529609,  0.15381187,  0.28613517, -0.11510384, -0.25761494,
       -0.07403318, -0.04513855,  0.24798553,  0.06555474,  0.14035505,
        0.00971497, -0.03118841,  0.32835603, -0.03530207, -0.00381901,
        0.0038671 ,  0.03232258,  0.0193231 , -0.01718037,  0.12143018,
       -0.10916604,  0.16562675, -0.08725581, -0.0136482 ,  0.05917208,
        0.11594134, -0.05374424, -0.18197812,  0.13060941,  0.32692796,
       -0.00556416, -0.03042129, -0.16028203, -0.02597837,  0.02022796],
      dtype=float32)
```

In [12]:
```
sims = model.wv.most_similar('Toyota Camry', topn=10)
sims
```

Out[12]:
```
[('Mazda 6', 0.9814350008964539),
 ('Nissan Altima', 0.97865229845047),
 ('Dodge Dynasty', 0.9780991673469543),
 ('Suzuki Aerio', 0.9769200682640076),
 ('Chevrolet Cruze', 0.9751349091529846),
 ('Pontiac Grand Am', 0.973942220211029),
 ('Ford Windstar', 0.9735517501831055),
 ('Kia Optima', 0.972167432308197),
 ('Oldsmobile Eighty-Eight Royale', 0.9713041186332703),
 ('Oldsmobile Cutlass Ciera', 0.9701311588287354)]
```

Import libraries and load data

In [1]:
```
#Importing Libraries

import pickle
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import chi2
import numpy as np
```

In [2]:
```
#Accessing document uploaded

path_df = "News_dataset.pickle"

with open(path_df, 'rb') as data:
    df = pickle.load(data)
```

In [3]:
```
#checking data

df.head()
```

Out[3]:

	File_Name	Content	Category	Complete_Filename	id	News_length
0	001.txt	Ad sales boost Time Warner profit\r\n\r\nQuart...	business	001.txt-business	1	2569
1	002.txt	Dollar gains on Greenspan speech\r\n\r\nThe do...	business	002.txt-business	1	2257
2	003.txt	Yukos unit buyer faces loan claim\r\n\r\nThe o...	business	003.txt-business	1	1557
3	004.txt	High fuel prices hit BA's profits\r\n\r\nBriti...	business	004.txt-business	1	2421
4	005.txt	Pernod takeover talk lifts Domecq\r\n\r\nShare...	business	005.txt-business	1	1575

In [4]:
```
#Chcking article

df.loc[1]['Content']
```

Out[4]:
```
'Dollar gains on Greenspan speech\r\n\r\nThe dollar has hit its highest level
against the euro in almost three months after the Federal Reserve head said t
he US trade deficit is set to stabilise.\r\n\r\nAnd Alan Greenspan highlighte
d the US government\'s willingness to curb spending and rising household savi
ngs as factors which may help to reduce it. In late trading in New York, the
dollar reached $1.2871 against the euro, from $1.2974 on Thursday. Market con
cerns about the deficit has hit the greenback in recent months. On Friday, Fe
deral Reserve chairman Mr Greenspan\'s speech in London ahead of the meeting
of G7 finance ministers sent the dollar higher after it had earlier tumbled o
n the back of worse-than-expected US jobs data. "I think the chairman\'s taki
ng a much more sanguine view on the current account deficit than he\'s taken
for some time," said Robert Sinche, head of currency strategy at Bank of Amer
ica in New York. "He\'s taking a longer-term view, laying out a set of condit
ions under which the current account deficit can improve this year and nex
t."\r\n\r\nWorries about the deficit concerns about China do, however, remai
n. China\'s currency remains pegged to the dollar and the US currency\'s shar
p falls in recent months have therefore made Chinese export prices highly com
petitive. But calls for a shift in Beijing\'s policy have fallen on deaf ear
s, despite recent comments in a major Chinese newspaper that the "time is rip
e" for a loosening of the peg. The G7 meeting is thought unlikely to produce
any meaningful movement in Chinese policy. In the meantime, the US Federal Re
serve\'s decision on 2 February to boost interest rates by a quarter of a poi
nt - the sixth such move in as many months - has opened up a differential wit
h European rates. The half-point window, some believe, could be enough to kee
p US assets looking more attractive, and could help prop up the dollar. The r
ecent falls have partly been the result of big budget deficits, as well as th
e US\'s yawning current account gap, both of which need to be funded by the b
uying of US bonds and assets by foreign firms and governments. The White Hous
e will announce its budget on Monday, and many commentators believe the defic
it will remain at close to half a trillion dollars.'
```

1. Text cleaning and preparation

In [5]:
```
#Text cleaning

df['Content_Parsed_1'] = df['Content'].str.replace("\r", " ")
df['Content_Parsed_1'] = df['Content_Parsed_1'].str.replace("\n", " ")
df['Content_Parsed_1'] = df['Content_Parsed_1'].str.replace("    ", " ")
df['Content_Parsed_1'] = df['Content_Parsed_1'].str.replace('"', '')
```

In [6]:
```
#Text preparation

df['Content_Parsed_2'] = df['Content_Parsed_1'].str.lower()        #all to lc

punctuation_signs = list("?:!.,;")                                  #remove pu
df['Content_Parsed_3'] = df['Content_Parsed_2']

for punct_sign in punctuation_signs:
    df['Content_Parsed_3'] = df['Content_Parsed_3'].str.replace(punct_sign, ''

df['Content_Parsed_4'] = df['Content_Parsed_3'].str.replace("'s", "")      #r
```

```
<ipython-input-6-3fcdc84e92bf>:9: FutureWarning: The default value of regex w
ill change from True to False in a future version. In addition, single charac
ter regular expressions will*not* be treated as literal strings when regex=Tr
ue.
  df['Content_Parsed_3'] = df['Content_Parsed_3'].str.replace(punct_sign, '')
```

a) Use any 1 method for Lemmatization

In [7]:
```
#Stemming and Lemmatization

nltk.download('punkt')
nltk.download('wordnet')

nltk.download('averaged_perceptron_tagger')
from nltk.corpus import wordnet
```

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

1st method for lemmatization

In [8]:
```
#Stemming and Lemmatization

wordnet_lemmatizer = WordNetLemmatizer()
nrows = len(df)
lemmatized_text_list = []

for row in range(0, nrows):

    # Create an empty list containing lemmatized words
    lemmatized_list = []

    # Save the text and its words into an object
    text = df.loc[row]['Content_Parsed_4']
    text_words = text.split(" ")

    # Iterate through every word to lemmatize
    for word in text_words:
        lemmatized_list.append(wordnet_lemmatizer.lemmatize(word, pos="v"))

    # Join the list
    lemmatized_text = " ".join(lemmatized_list)

    # Append to the list containing the texts
    lemmatized_text_list.append(lemmatized_text)

df['Content_Parsed_5'] = lemmatized_text_list
```

In [9]:
```
df['Content_Parsed_5']
```

Out[9]:
```
0       ad sales boost time warner profit quarterly pr...
1       dollar gain on greenspan speech the dollar hav...
2       yukos unit buyer face loan claim the owners of...
3       high fuel price hit ba profit british airways ...
4       pernod takeover talk lift domecq share in uk d...
                              ...
2220    bt program to beat dialler scam bt be introduc...
2221    spam e-mail tempt net shoppers computer users ...
2222    be careful how you code a new european directi...
2223    us cyber security chief resign the man make su...
2224    lose yourself in online game online role play ...
Name: Content_Parsed_5, Length: 2225, dtype: object
```

2nd method for lemmatization

In [10]:
```
lemmatizer = WordNetLemmatizer()

# function to convert nltk tag to wordnet tag
def nltk_tag_to_wordnet_tag(nltk_tag):
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

def lemmatize_sentence(sentence):
    #tokenize the sentence and find the POS tag for each token
    nltk_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
    #tuple of (token, wordnet_tag)
    wordnet_tagged = map(lambda x: (x[0], nltk_tag_to_wordnet_tag(x[1])), nltk
    lemmatized_sentence = []
    for word, tag in wordnet_tagged:
        if tag is None:
            #if there is no available tag, append the token as is
            lemmatized_sentence.append(word)
        else:
            #else use the tag to lemmatize the token
            lemmatized_sentence.append(lemmatizer.lemmatize(word, tag))
    return " ".join(lemmatized_sentence)

nrows = len(df)
lemmatized_text_list = []

for row in range(0, nrows):
    lemmatized_text = lemmatize_sentence(df.loc[row]['Content_Parsed_4'])
    lemmatized_text_list.append(lemmatized_text)

df['Content_Parsed_5'] = lemmatized_text_list
```

In [11]:
```
df['Content_Parsed_5']
```

Out[11]:
```
0       ad sale boost time warner profit quarterly pro...
1       dollar gain on greenspan speech the dollar hav...
2       yukos unit buyer face loan claim the owner of ...
3       high fuel price hit ba profit british airway h...
4       pernod takeover talk lift domecq share in uk d...
                              ...
2220    bt program to beat dialler scam bt be introduc...
2221    spam e-mails tempt net shopper computer user a...
2222    be careful how you code a new european directi...
2223    us cyber security chief resign the man make su...
2224    lose yourself in online gaming online role pla...
Name: Content_Parsed_5, Length: 2225, dtype: object
```

b) Use any 1 method for stop word

In [12]:
```
#Downloading

nltk.download('stopwords')
```

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

Out[12]:
```
True
```

In [13]:
```
#Removing stop words

stop_words = list(stopwords.words('english'))
```

1st Method

In [14]:
```

df['Content_Parsed_6'] = df['Content_Parsed_5']

for stop_word in stop_words:

    regex_stopword = r"\b" + stop_word + r"\b"
    df['Content_Parsed_6'] = df['Content_Parsed_6'].str.replace(regex_stopword
```

```
<ipython-input-14-3b7196a1b53b>:6: FutureWarning: The default value of regex
will change from True to False in a future version.
  df['Content_Parsed_6'] = df['Content_Parsed_6'].str.replace(regex_stopword,
'')
```

In [15]:
```
df.loc[5]['Content_Parsed_6']
```

Out[15]:
```
'japan narrowly escape recession japan economy teeter   brink   technical rec
ession   three month   september figure show revised figure indicate growth
01 % -   similar-sized contraction   previous quarter   annual basis  data su
ggest annual growth   02 % suggest  much  hesitant recovery   previously  thi
nk  common technical definition   recession  two successive quarter  negative
growth  government  keen  play   worrying implication   data  maintain  view
japan economy remain   minor adjustment phase   upward climb    monitor devel
opment carefully say economy minister heizo takenaka     face   strengthen yen
make export less competitive  indication  weaken economic condition ahead obs
erver  less sanguine  paint  picture   recovery much patchy  previously think
say paul sheard economist  lehman brother  tokyo improvement   job market app
arently  yet  fee   domestic demand  private consumption   02 %   third quart
er'
```

2nd Method

In [16]:
```
stop_list_final=[]
nrows = len(df)
stopwords_english = stopwords.words('english')

for row in range(0, nrows):

    # Create an empty list containing no stop words
    stop_list = []

    # Save the text and its words into an object
    text = df.loc[row]['Content_Parsed_5']
    text_words = text.split(" ")

    # Iterate through every word to remove stopwords
    for word in text_words:
        if (word not in stopwords_english):
          stop_list.append(word)

    # Join the list
    stop_text = " ".join(stop_list)

    # Append to the list containing the texts
    stop_list_final.append(stop_text)

df['Content_Parsed_6'] = stop_list_final
```

In [17]:
```
df.loc[5]['Content_Parsed_6']
```

Out[17]:
```
'japan narrowly escape recession japan economy teeter brink technical recessi
on three month september figure show revised figure indicate growth 01 % - si
milar-sized contraction previous quarter annual basis data suggest annual gro
wth 02 % suggest much hesitant recovery previously think common technical def
inition recession two successive quarter negative growth government keen play
worrying implication data maintain view japan economy remain minor adjustment
phase upward climb monitor development carefully say economy minister heizo t
akenaka face strengthen yen make export less competitive indication weaken ec
onomic condition ahead observer less sanguine paint picture recovery much pat
chy previously think say paul sheard economist lehman brother tokyo improveme
nt job market apparently yet fee domestic demand private consumption 02 % thi
rd quarter'
```

In [18]:
```
#Checking data

df.head(1)
```

Out[18]:

	File_Name	Content	Category	Complete_Filename	id	News_length	Content_Parsed
0	001.txt	Ad sales boost Time Warner profit\r\n\r\nQuart...	business	001.txt-business	1	2569	Ad sales bo Time Warner pr Quarterly p

In [19]:
```
#Removing the old content_parsed columns

list_columns = ["File_Name", "Category", "Complete_Filename", "Content", "Cont
df = df[list_columns]

df = df.rename(columns={'Content_Parsed_6': 'Content_Parsed'})
```

In [20]:
```
df.head()
```

Out[20]:

	File_Name	Category	Complete_Filename	Content	Content_Parsed
0	001.txt	business	001.txt-business	Ad sales boost Time Warner profit\r\n\r\nQuart...	ad sale boost time warner profit quarterly pro...
1	002.txt	business	002.txt-business	Dollar gains on Greenspan speech\r\n\r\nThe do...	dollar gain greenspan speech dollar hit high l...
2	003.txt	business	003.txt-business	Yukos unit buyer faces loan claim\r\n\r\nThe o...	yukos unit buyer face loan claim owner embattl...
3	004.txt	business	004.txt-business	High fuel prices hit BA's profits\r\n\r\nBriti...	high fuel price hit ba profit british airway b...
4	005.txt	business	005.txt-business	Pernod takeover talk lifts Domecq\r\n\r\nShare...	pernod takeover talk lift domecq share uk drin...

2. Label coding

In [21]:
```
#Generating new column for Category codes

category_codes = {
    'business': 0,
    'entertainment': 1,
    'politics': 2,
    'sport': 3,
    'tech': 4
}

# Category mapping
df['Category_Code'] = df['Category']
df = df.replace({'Category_Code':category_codes})
```

In [22]:
```
df.head()
```

Out[22]:

	File_Name	Category	Complete_Filename	Content	Content_Parsed	Category_Co
0	001.txt	business	001.txt-business	Ad sales boost Time Warner profit\r\n\r\nQuart...	ad sale boost time warner profit quarterly pro...	
1	002.txt	business	002.txt-business	Dollar gains on Greenspan speech\r\n\r\nThe do...	dollar gain greenspan speech dollar hit high l...	
2	003.txt	business	003.txt-business	Yukos unit buyer faces loan claim\r\n\r\nThe o...	yukos unit buyer face loan claim owner embattl...	
3	004.txt	business	004.txt-business	High fuel prices hit BA's profits\r\n\r\nBriti...	high fuel price hit ba profit british airway b...	
4	005.txt	business	005.txt-business	Pernod takeover talk lifts Domecq\r\n\r\nShare...	pernod takeover talk lift domecq share uk drin...	

3. Train - test split

In [23]:
```
X_train, X_test, y_train, y_test = train_test_split(df['Content_Parsed'],
                                                    df['Category_Code'],
                                                    test_size=0.15,
                                                    random_state=8)
```

4. Text representation

TF-IDF Vectors

unigrams & bigrams corresponding to a particular category

In [24]:
```
# Parameter election
ngram_range = (1,2)
min_df = 10
max_df = 1.
max_features = 300
```

In [25]:
```
tfidf = TfidfVectorizer(encoding='utf-8',
                        ngram_range=ngram_range,
                        stop_words=None,
                        lowercase=False,
                        max_df=max_df,
                        min_df=min_df,
                        max_features=max_features,
                        norm='l2',
                        sublinear_tf=True)

features_train = tfidf.fit_transform(X_train).toarray()
labels_train = y_train
print(features_train.shape)

features_test = tfidf.transform(X_test).toarray()
labels_test = y_test
print(features_test.shape)
```

```
(1891, 300)
(334, 300)
```

In [26]:
```
from sklearn.feature_selection import chi2
import numpy as np

for Product, category_id in sorted(category_codes.items()):
    features_chi2 = chi2(features_train, labels_train == category_id)
    indices = np.argsort(features_chi2[0])
    feature_names = np.array(tfidf.get_feature_names())[indices]
    unigrams = [v for v in feature_names if len(v.split(' ')) == 1]
    bigrams = [v for v in feature_names if len(v.split(' ')) == 2]
    print("# '{}' category:".format(Product))
    print("  . Most correlated unigrams:\n. {}".format('\n. '.join(unigrams[-5
    print("  . Most correlated bigrams:\n. {}".format('\n. '.join(bigrams[-2:]
    print("")
```

```
# 'business' category:
  . Most correlated unigrams:
. price
. market
. economy
. growth
. bank
  . Most correlated bigrams:
. last year
. year old

# 'entertainment' category:
  . Most correlated unigrams:
. best
. music
. star
. award
. film
  . Most correlated bigrams:
. mr blair
. prime minister

# 'politics' category:
  . Most correlated unigrams:
. blair
. party
. election
. tory
. labour
  . Most correlated bigrams:
. prime minister
. mr blair

# 'sport' category:
  . Most correlated unigrams:
. side
. player
. team
. game
. match
  . Most correlated bigrams:
. say mr
. year old

# 'tech' category:
  . Most correlated unigrams:
. mobile
. software
. technology
. computer
. user
  . Most correlated bigrams:
. year old
. say mr
```

In [27]:
```
bigrams
```

Out[27]:
```
['tell bbc', 'last year', 'mr blair', 'prime minister', 'year old', 'say mr']
```

Unigrams are more relevnat to the category as compared with bigrams

In [1]:
```
import torch.nn as nn
import torch
import torch.nn.functional as F
import math,copy,re
import warnings
import pandas as pd
import numpy as np
import seaborn as sns
import torchtext
import matplotlib.pyplot as plt
warnings.simplefilter("ignore")
print(torch.__version__)
```

```
1.11.0+cpu
```

In [2]:
```
class Embedding(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        """
        Args:
            vocab_size: size of vocabulary
            embed_dim: dimension of embeddings
        """
        super(Embedding, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            out: embedding vector
        """
        out = self.embed(x)
        return out
```

In [3]:
```
class PositionalEmbedding(nn.Module):
    def __init__(self,max_seq_len,embed_model_dim):
        """
        Args:
            seq_len: length of input sequence
            embed_model_dim: demension of embedding
        """
        super(PositionalEmbedding, self).__init__()
        self.embed_dim = embed_model_dim

        pe = torch.zeros(max_seq_len,self.embed_dim)
        for pos in range(max_seq_len):
            for i in range(0,self.embed_dim,2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/self.embed_dim)
                pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/self.
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            x: output
        """
        x = x * math.sqrt(self.embed_dim)
        #add constant to embedding
        seq_len = x.size(1)
        x = x + torch.autograd.Variable(self.pe[:,:seq_len], requires_grad=Fal
        return x
```

In [4]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim=512, n_heads=8):
        """
        Args:
            embed_dim: dimension of embeding vector output
            n_heads: number of self attention heads
        """
        super(MultiHeadAttention, self).__init__()

        self.embed_dim = embed_dim    #512 dim
        self.n_heads = n_heads   #8
        self.single_head_dim = int(self.embed_dim / self.n_heads)
        self.query_matrix = nn.Linear(self.single_head_dim , self.single_head_
        self.key_matrix = nn.Linear(self.single_head_dim  , self.single_head_d
        self.value_matrix = nn.Linear(self.single_head_dim ,self.single_head_d
        self.out = nn.Linear(self.n_heads*self.single_head_dim ,self.embed_dim

    def forward(self,key,query,value,mask=None):

        """
        Args:
           key : key vector
           query : query vector
           value : value vector
           mask: mask for decoder

        Returns:
           output vector from multihead attention
        """
        batch_size = key.size(0)
        seq_length = key.size(1)
        seq_length_query = query.size(1)

        # 32x10x512
        key = key.view(batch_size, seq_length, self.n_heads, self.single_head_
        query = query.view(batch_size, seq_length_query, self.n_heads, self.si
        value = value.view(batch_size, seq_length, self.n_heads, self.single_h

        k = self.key_matrix(key)       # (32x10x8x64)
        q = self.query_matrix(query)
        v = self.value_matrix(value)
        q = q.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim
        k = k.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim
        v = v.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim

        # computes attention
        # adjust key for matrix multiplication
        k_adjusted = k.transpose(-1,-2)  #(batch_size, n_heads, single_head_di
        product = torch.matmul(q, k_adjusted)  #(32 x 8 x 10 x 64) x (32 x 8 x
        if mask is not None:
             product = product.masked_fill(mask == 0, float("-1e20"))

        #divising by square root of key dimension
        product = product / math.sqrt(self.single_head_dim) # / sqrt(64)

        #applying softmax
        scores = F.softmax(product, dim=-1)
```

```
        #mutiply with value matrix
        scores = torch.matmul(scores, v)  ##(32x8x 10x 10) x (32 x 8 x 10 x 64
        concat = scores.transpose(1,2).contiguous().view(batch_size, seq_lengt

        output = self.out(concat) #(32,10,512) -> (32,10,512)

        return output
```

In [5]:
```
class TransformerBlock(nn.Module):
  def __init__(self, embed_dim, expansion_factor=4, n_heads=8):
    super(TransformerBlock, self).__init__()
    self.attention = MultiHeadAttention(embed_dim, n_heads)
    self.norm1 = nn.LayerNorm(embed_dim)
    self.norm2 = nn.LayerNorm(embed_dim)
    self.feed_forward = nn.Sequential(nn.Linear(embed_dim, expansion_factor*em
    self.dropout1 = nn.Dropout(0.2)
    self.dropout2 = nn.Dropout(0.2)

  def forward(self,key,query,value):
    attention_out = self.attention(key,query,value)  #32x10x512
    attention_residual_out = attention_out + value  #32x10x512
    norm1_out = self.dropout1(self.norm1(attention_residual_out)) #32x10x512
    feed_fwd_out = self.feed_forward(norm1_out) #32x10x512 -> #32x10x2048 -> 3
    feed_fwd_residual_out = feed_fwd_out + norm1_out #32x10x512
    norm2_out = self.dropout2(self.norm2(feed_fwd_residual_out)) #32x10x512
    return norm2_out


class TransformerEncoder(nn.Module):
  def __init__(self, seq_len, vocab_size, embed_dim, num_layers=2, expansion_f
    super(TransformerEncoder, self).__init__()
    self.embedding_layer = Embedding(vocab_size, embed_dim)
    self.positional_encoder = PositionalEmbedding(seq_len, embed_dim)
    self.layers = nn.ModuleList([TransformerBlock(embed_dim, expansion_factor,

  def forward(self, x):
    embed_out = self.embedding_layer(x)
    out = self.positional_encoder(embed_out)
    for layer in self.layers:
      out = layer(out,out,out)
    return out
```

In [6]:
```
class DecoderBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=8):
      super(DecoderBlock, self).__init__()
      self.attention = MultiHeadAttention(embed_dim, n_heads=8)
      self.norm = nn.LayerNorm(embed_dim)
      self.dropout = nn.Dropout(0.2)
      self.transformer_block = TransformerBlock(embed_dim, expansion_factor, r
    def forward(self, key, query, x,mask):
      attention = self.attention(x,x,x,mask=mask) #32x10x512
      value = self.dropout(self.norm(attention + x))
      out = self.transformer_block(key, query, value)
      return out

class TransformerDecoder(nn.Module):
  def __init__(self, target_vocab_size, embed_dim, seq_len, num_layers=2, expa
    super(TransformerDecoder, self).__init__()
    self.word_embedding = nn.Embedding(target_vocab_size, embed_dim)
    self.position_embedding = PositionalEmbedding(seq_len, embed_dim)
    self.layers = nn.ModuleList([DecoderBlock(embed_dim, expansion_factor=4, r
    self.fc_out = nn.Linear(embed_dim, target_vocab_size)
    self.dropout = nn.Dropout(0.2)
  def forward(self, x, enc_out, mask):
    x = self.word_embedding(x)  #32x10x512
    x = self.position_embedding(x) #32x10x512
    x = self.dropout(x)

    for layer in self.layers:
      x = layer(enc_out, x, enc_out, mask)
      out = F.softmax(self.fc_out(x))
    return out
```

In [7]:
```
class Transformer(nn.Module):
    def __init__(self, embed_dim, src_vocab_size, target_vocab_size, seq_lengt
        super(Transformer, self).__init__()
        self.target_vocab_size = target_vocab_size

        self.encoder = TransformerEncoder(seq_length, src_vocab_size, embed_di
        self.decoder = TransformerDecoder(target_vocab_size, embed_dim, seq_le
    def make_trg_mask(self, trg):
      batch_size, trg_len = trg.shape
      trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(batch_size,
      return trg_mask
    def decode(self,src,trg):
      trg_mask = self.make_trg_mask(trg)
      enc_out = self.encoder(src)
      out_labels = []
      batch_size,seq_len = src.shape[0],src.shape[1]
      out = trg
      for i in range(seq_len):
        out = self.decoder(out,enc_out,trg_mask) #bs x seq_len x vocab_dim
            # taking the last token
        out = out[:,-1,:]

        out = out.argmax(-1)
        out_labels.append(out.item())
        out = torch.unsqueeze(out,axis=0)
      return out_labels
    def forward(self, src, trg):
      trg_mask = self.make_trg_mask(trg)
      enc_out = self.encoder(src)

      outputs = self.decoder(trg, enc_out, trg_mask)
      return outputs
```

In [8]:
```
src_vocab_size = 11
target_vocab_size = 11
num_layers = 6
seq_length= 12


# let 0 be sos token and 1 be eos token
src = torch.tensor([[0, 2, 5, 6, 4, 3, 9, 5, 2, 9, 10, 1],
                    [0, 2, 8, 7, 3, 4, 5, 6, 7, 2, 10, 1]])
target = torch.tensor([[0, 1, 7, 4, 3, 5, 9, 2, 8, 10, 9, 1],
                       [0, 1, 5, 6, 2, 4, 7, 6, 2, 8, 10, 1]])

print(src.shape,target.shape)
model = Transformer(embed_dim=512, src_vocab_size=src_vocab_size,
                    target_vocab_size=target_vocab_size, seq_length=seq_length
                    num_layers=num_layers, expansion_factor=4, n_heads=8)
model
```

```
torch.Size([2, 12]) torch.Size([2, 12])
```

Out[8]:
```
Transformer(
  (encoder): TransformerEncoder(
    (embedding_layer): Embedding(
      (embed): Embedding(11, 512)
    )
    (positional_encoder): PositionalEmbedding()
    (layers): ModuleList(
      (0): TransformerBlock(
        (attention): MultiHeadAttention(
          (query_matrix): Linear(in_features=64, out_features=64, bias=Fal
se)
          (key_matrix): Linear(in_features=64, out_features=64, bias=Fals
e)
          (value_matrix): Linear(in_features=64, out_features=64, bias=Fal
se)
          (out): Linear(in_features=512, out_features=512, bias=True)
        )
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
```

In [9]:
```
out = model(src, target)
out.shape
```

Out[9]:
```
torch.Size([2, 12, 11])
```

In [10]:
```
model = Transformer(embed_dim=512, src_vocab_size=src_vocab_size,
                    target_vocab_size=target_vocab_size, seq_length=seq_length
                    num_layers=num_layers, expansion_factor=4, n_heads=8)



src = torch.tensor([[0, 2, 5, 6, 4, 3, 9, 5, 2, 9, 10, 1]])
trg = torch.tensor([[0]])
print(src.shape,trg.shape)
out = model.decode(src, trg)
out
```

```
torch.Size([1, 12]) torch.Size([1, 1])
```

Out[10]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```