EXPERIMENT 7

TITLE: Text Analytics

PROBLEM STATEMENT: -

1. Extract Sample document and apply following document preprocessing methods: Tokenization, POS Tagging, stop words removal, Stemming and Lemmatization.
2. Create representation of document by calculating Term Frequency and Inverse Document Frequency.

OBJECTIVE: Learn how to create and develop sentiment analysis using Python.

PREREQUISITE:-

1 Basic of Python Programming
2 Tokenization, POS Tagging, stop words removal, Stemming and Lemmatization.

THEORY:

- **Text Analytics and NLP**

Text communication is one of the most popular forms of day to day conversion. We chat, message, tweet, share status, email, write blogs, share opinions and feedback in our daily routine. All of these activities are generating text in a significant amount, which is unstructured in nature. In this area of the online marketplace and social media, It is essential to analyze vast quantities of data, to understand people's opinion.

NLP is applicable in several problems from speech recognition, language translation, classifying documents to information extraction. Analyzing movie reviews is one of the classic examples to demonstrate a simple NLP Bag-of-words model, on movie reviews.

-
-
-

○

○ **Compare Text Analytics, NLP and Text Mining**

Text mining is also referred to as text analytics. Text mining is a process of exploring sizable textual data and finding patterns. Text Mining processes the text itself, while NLP processes with the underlying metadata. Finding frequency counts of words, length of the sentence, presence/absence of specific words is known as text mining. Natural language processing is one of the components of text mining. NLP helps identify sentiment, finding entities in the sentence, and category of blog/article. Text mining is preprocessed data for text analytics. In Text Analytics, statistical and machine learning algorithms are used to classify information.

■ **Text Analysis Operations using NLTK**

NLTK is a powerful Python package that provides a set of diverse natural language algorithms. It is free, opensource, easy to use, large community, and well documented. NLTK consists of the most common algorithms such as tokenizing, part-of-speech tagging, stemming, sentiment analysis, topic segmentation, and named entity recognition. NLTK helps the computer to analyze, preprocess, and understand the written text.

NLTK stands for natural language tool kit

```
!pip install nltk
```

```
Requirement already satisfied: nltk in /home/northout/anaconda2/lib/python2.7/site-packages
Requirement already satisfied: six in /home/northout/anaconda2/lib/python2.7/site-packages (from nltk)

[33mYou are using pip version 9.0.1, however version 10.0.1 is available.

You should consider upgrading via the 'pip install --upgrade pip' command.[0m
```

```
#Loading NLTK

import nltk
```

■ Tokenization

Tokenization is the first step in text analytics. The process of breaking down a text paragraph into smaller chunks such as words or sentences is called Tokenization. Token is a single entity that is the building blocks for a sentence or paragraph.

Sentence Tokenization

Sentence tokenizer breaks text paragraph into sentences.

```
from nltk.tokenize import sent_tokenize

text="""Hello Mr. Smith, how are you doing today? The weather is great, and city is awesome.

The sky is pinkish-blue. You shouldn't eat cardboard"""

tokenized_text=sent_tokenize(text)

print(tokenized_text)
```

OUTPUT:

```
['Hello Mr. Smith, how are you doing today?', 'The weather is great, and city is awesome.', 'The sky is pinkish-blue.',
"You shouldn't eat cardboard"]
```

Here, the given text is tokenized into sentences.

Word Tokenization

Word tokenizer breaks text paragraphs into words.

```
from nltk.tokenize import word_tokenize

tokenized_word=word_tokenize(text)

print(tokenized_word)
```

OUTPUT:

```
['Hello', 'Mr.', 'Smith', ',', 'how', 'are', 'you', 'doing', 'today', '?', 'The', 'weather', 'is', 'great', ',', 'and', 'city', 'is', 'awesome',
'.', 'The', 'sky', 'is', 'pinkish-blue', '.', 'You', 'should', "n't", 'eat', 'cardboard']
```

## Frequency Distribution

```
from nltk.probability import FreqDist

fdist = FreqDist(tokenized_word)

print(fdist)
```

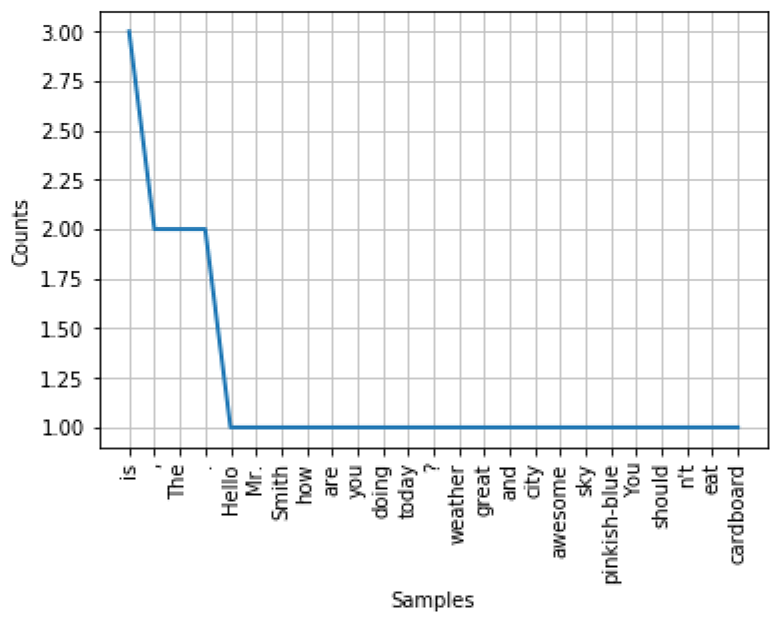
```
<FreqDist with 25 samples and 30 outcomes>
```

```
fdist.most_common(2)
```

```
[('is', 3), (',', 2)]
```

# Frequency Distribution Plot

```
import matplotlib.pyplot as plt

fdist.plot(30,cumulative=False)

plt.show()
```



■

■

■

- **Stopwords**

Stopwords considered as noise in the text. Text may contain stop words such as is, am, are, this, a, an, the, etc.

In NLTK for removing stopwords, you need to create a list of stopwords and filter out your list of tokens from these words.

```
from nltk.corpus import stopwords
```

```
stop_words=set(stopwords.words("english"))

print(stop_words)
```

```
{'their', 'then', 'not', 'ma', 'here', 'other', 'won', 'up', 'weren', 'being', 'we', 'those', 'an', 'them', 'which', 'him', 'so',
'yourselves', 'what', 'own', 'has', 'should', 'above', 'in', 'myself', 'against', 'that', 'before', 't', 'just', 'into', 'about', 'most',
'd', 'where', 'our', 'or', 'such', 'ours', 'of', 'doesn', 'further', 'needn', 'now', 'some', 'too', 'hasn', 'more', 'the', 'yours', 'her',
'below', 'same', 'how', 'very', 'is', 'did', 'you', 'his', 'when', 'few', 'does', 'down', 'yourself', 'i', 'do', 'both', 'shan', 'have',
'itself', 'shouldn', 'through', 'themselves', 'o', 'didn', 've', 'm', 'off', 'out', 'but', 'and', 'doing', 'any', 'nor', 'over', 'had',
'because', 'himself', 'theirs', 'me', 'by', 'she', 'whom', 'hers', 're', 'hadn', 'who', 'he', 'my', 'if', 'will', 'are', 'why', 'from', 'am',
'with', 'been', 'its', 'ourselves', 'ain', 'couldn', 'a', 'aren', 'under', 'll', 'on', 'y', 'can', 'they', 'than', 'after', 'wouldn', 'each',
'once', 'mightn', 'for', 'this', 'these', 's', 'only', 'haven', 'having', 'all', 'don', 'it', 'there', 'until', 'again', 'to', 'while', 'be', 'no',
'during', 'herself', 'as', 'mustn', 'between', 'was', 'at', 'your', 'were', 'isn', 'wasn'}
```

Removing Stopwords

```
filtered_sent=[]

for w in tokenized_sent:

  if w not in stop_words:

    filtered_sent.append(w)

print("Tokenized Sentence:",tokenized_sent)

print("Filtered Sentence:",filtered_sent)
```

```
Tokenized Sentence: ['Hello', 'Mr.', 'Smith', ',', 'how', 'are', 'you', 'doing', 'today', '?']

Filtered Sentence: ['Hello', 'Mr.', 'Smith', ',', 'today', '?']
```

■ Lexicon Normalization

Lexicon normalization considers another type of noise in the text. For example, connection, connected, connecting word reduce to a common word "connect". It reduces derivationally related forms of a word to a common root word.

Stemming

Stemming is a process of linguistic normalization, which reduces words to their root word or chops off the derivational affixes. For example, connection, connected, connecting word reduce to a common word "connect".

```
# Stemming

from nltk.stem import PorterStemmer

from nltk.tokenize import sent_tokenize, word_tokenize



ps = PorterStemmer()



stemmed_words=[]

for w in filtered_sent:

    stemmed_words.append(ps.stem(w))
```

```
print("Filtered Sentence:",filtered_sent)

print("Stemmed Sentence:",stemmed_words)
```

```
Filtered Sentence: ['Hello', 'Mr.', 'Smith', ',', 'today', '?']

Stemmed Sentence: ['hello', 'mr.', 'smith', ',', 'today', '?']
```

Lemmatization

Lemmatization reduces words to their base word, which is linguistically correct lemmas. It transforms root words with the use of vocabulary and morphological analysis. Lemmatization is usually more sophisticated than stemming. Stemmer works on an individual word without knowledge of the context. For example, The word "better" has "good" as its lemma. This thing will miss by stemming because it requires a dictionary look-up.

```
#Lexicon Normalization

#performing stemming and Lemmatization


from nltk.stem.wordnet import WordNetLemmatizer

lem = WordNetLemmatizer()
```

```
from nltk.stem.porter import PorterStemmer

stem = PorterStemmer()


word = "flying"

print("Lemmatized Word:",lem.lemmatize(word,"v"))

print("Stemmed Word:",stem.stem(word))
```

```
Lemmatized Word: fly

Stemmed Word: fli
```

■

■

■

■ POS Tagging

The primary target of Part-of-Speech(POS) tagging is to identify the grammatical group of a given word. Whether it is a NOUN, PRONOUN, ADJECTIVE, VERB, ADVERBS, etc. based on the context. POS Tagging looks for relationships within the sentence and assigns a corresponding tag to the word.

```
sent = "Albert Einstein was born in Ulm, Germany in 1879."
```

```
tokens=nltk.word_tokenize(sent)

print(tokens)
```

```
['Albert', 'Einstein', 'was', 'born', 'in', 'Ulm', ',', 'Germany', 'in', '1879', '.']
```

```
nltk.pos_tag(tokens)
```

```
[('Albert', 'NNP'),
 ('Einstein', 'NNP'),
 ('was', 'VBD'),
 ('born', 'VBN'),
 ('in', 'IN'),
 ('Ulm', 'NNP'),
 (',', ','),
 ('Germany', 'NNP'),
 ('in', 'IN'),
 ('1879', 'CD'),
 ('.', '.')]
```

2. Create representation of document by calculating Term Frequency and Inverse Document Frequency.

Creating TF-IDF Model

This technique is used to find meaning of sentences consisting of words and cancels out the incapabilities of Bag of Words technique which is good for text classification or for helping a machine read words in numbers.

1 - Terminology :

- t — term (word)
- d — document (set of words)
- N — count of corpus
- corpus — the total document set

2 -Term Frequency (TF):

Suppose we have a set of English text documents and wish to rank which document is most relevant to the query , "Data Science is awesome !" A simple way to start out is by eliminating documents that do not contain all three words "Data","is", "Science", and "awesome", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document; the number of times a term occurs in a document is called its *term frequency*. ***The weight of a term that occurs in a document is simply proportional to the term frequency.***

Formula :

tf(t,d) = count of t in d / number of words in d

3 -Document Frequency :

This measures the importance of documents in the whole set of corpus, this is very similar to TF. The only difference is that TF is frequency counter for a term t in document d, whereas DF is the count of **occurrences** of term t in the document set N. In other words, DF is the number of documents in which the word is present. We consider one occurrence if the term consists in the document at least once, we do not need to know the number of times the term is present.

df(t) = occurrence of t in documents

4 -Inverse Document Frequency(IDF):

While computing TF, all terms are considered equally important. However it is known that certain terms, such as “is”, “of”, and “that”, may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scaling up the rare ones, by computing IDF, an *inverse document frequency* factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

IDF is the inverse of the document frequency which measures the informativeness of term t. When we calculate IDF, it will be very low for the most occurring words such as stop words (because stop words such as “is” is present in almost all of the documents, and N/df will give a very low value to that word). This finally gives what we want, a relative weightage.

$$idf(t) = N/df$$

Now there are few other problems with the IDF , in case of a large corpus,say 100,000,000 , the IDF value explodes , to avoid the effect we take the log of idf .

During the query time, when a word which is not in vocab occurs, the df will be 0. As we cannot divide by 0, we smoothen the value by adding 1 to the denominator.

that’s the final formula:

Formula :

idf(t) = log(N/(df + 1))

tf-idf now is the right measure to evaluate how important a word is to a document in a collection or corpus.here are many different variations of TF-IDF but for now let us concentrate on this basic version.

Formula :

tf-idf(t, d) = tf(t, d) * log(N/(df + 1))

5 -Implementing TF-IDF in Python From Scratch :

To make TF-IDF from scratch in python,let's imagine those two sentences from different document :

first_sentence : "Data Science is the sexiest job of the 21st century".

second_sentence : "machine learning is the key for data science".

First step we have to create the TF function to calculate total word frequency for all documents. Here are the codes below:

first as usual we should import the necessary libraries :

```
import pandas as pd
```

```
import sklearn as sk
```

```
import math
```

so let’s load our sentences and combine them together in a single set :

```
first_sentence = "Data Science is the sexiest job of the 21st
century"
```

```
second_sentence = "machine learning is the key for data science"
```

```
#split so each word have their own string
```

```
first_sentence = first_sentence.split(" ")
```

```
second_sentence = second_sentence.split(" ")#join them to remove common duplicate words
```

```
total= set(first_sentence).union(set(second_sentence))
```

```
print(total)
```

Output :

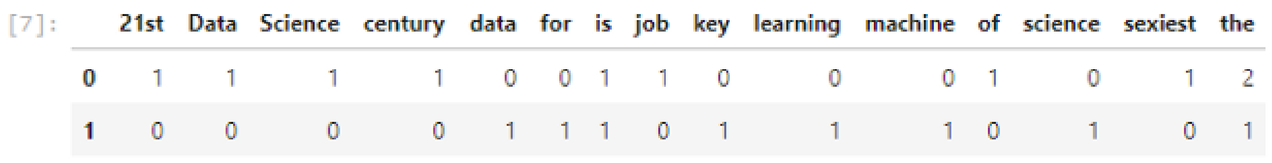
```
{'data', 'Science', 'job', 'sexiest', 'the', 'for', 'science', 'machine', 'of', 'is', 'learning', '21st', 'key', 'Data', 'century'}
```

Now lets add a way to count the words using a dictionary key-value pairing for both sentences :

```
wordDictA = dict.fromkeys(total, 0)

wordDictB = dict.fromkeys(total, 0)

for word in first_sentence:

   wordDictA[word]+=1


for word in second_sentence:

   wordDictB[word]+=1
```

Now we put them in a dataframe and then view the result:

```
pd.DataFrame([wordDictA, wordDictB])
```



[7]:

	21st	Data	Science	century	data	for	is	job	key	learning	machine	of	science	sexiest	the
0	1	1	1	1	0	0	1	1	0	0	0	1	0	1	2
1	0	0	0	0	1	1	1	0	1	1	1	0	1	0	1

No let's writing the TF Function :

```
def computeTF(wordDict, doc):

   tfDict = {}

   corpusCount = len(doc)

   for word, count in wordDict.items():

       tfDict[word] = count/float(corpusCount)

   return(tfDict)
```

```
#running our sentences through the tf function:
```

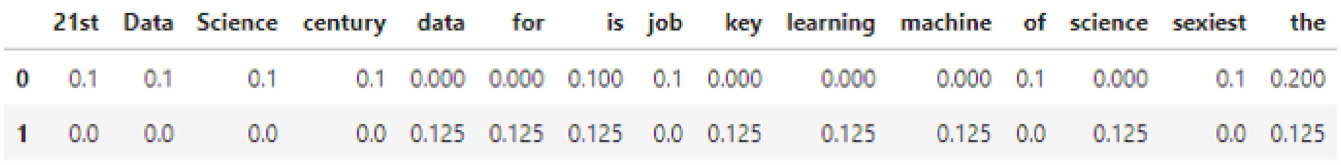
```
tfFirst = computeTF(wordDictA, first_sentence)
```

```
tfSecond = computeTF(wordDictB, second_sentence)
```

```
#Converting to dataframe for visualization
```

```
tf = pd.DataFrame([tfFirst, tfSecond])
```

and this is the expected output :

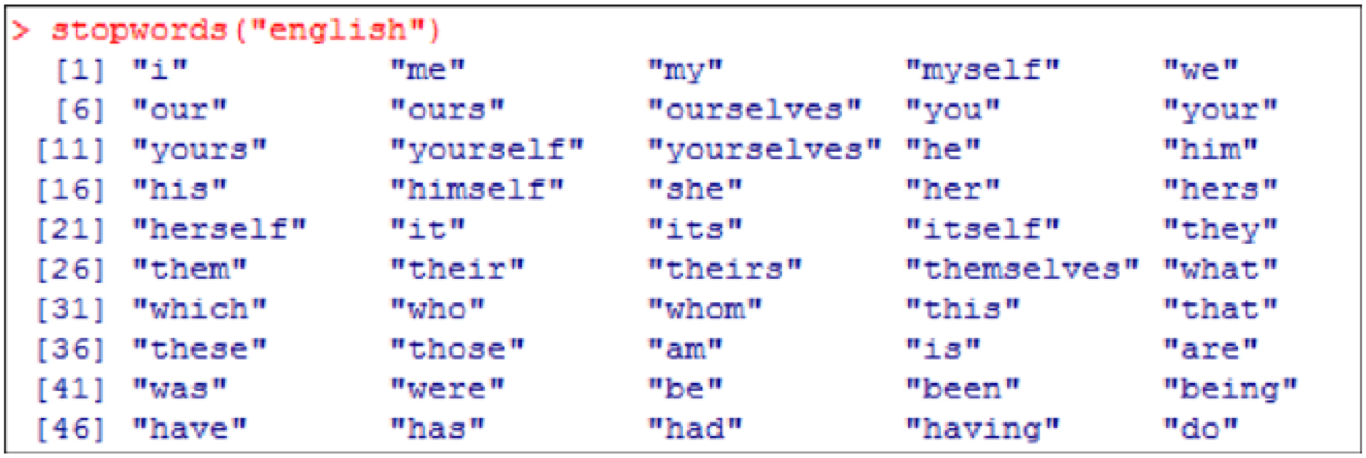


	21st	Data	Science	century	data	for	is	job	key	learning	machine	of	science	sexiest	the
0	0.1	0.1	0.1	0.1	0.000	0.000	0.100	0.1	0.000	0.000	0.000	0.1	0.000	0.1	0.200
1	0.0	0.0	0.0	0.0	0.125	0.125	0.125	0.0	0.125	0.125	0.125	0.0	0.125	0.0	0.125

That’s all for the TF formula , just i wanna talk about stop words that we should eliminate because they are the most commonly occurring words which don’t give any additional value to the document vector .in-fact removing these will increase computation and space efficiency.

nltk library has a method to download the stopwords, so instead of explicitly mentioning all the stopwords ourselves we can just use the nltk library and iterate over all the words and remove the stop words. There are many efficient ways to do this, but i'll just give a simple method.

those a sample of a stopwords in english language :


```
> stopwords("english")
  [1] "i"          "me"         "my"         "myself"     "we"
  [6] "our"        "ours"       "ourselves"  "you"        "your"
 [11] "yours"      "yourself"   "yourselves" "he"         "him"
 [16] "his"        "himself"    "she"        "her"        "hers"
 [21] "herself"    "it"         "its"        "itself"     "they"
 [26] "them"       "their"      "theirs"     "themselves" "what"
 [31] "which"      "who"        "whom"       "this"       "that"
 [36] "these"      "those"      "am"         "is"         "are"
 [41] "was"        "were"       "be"         "been"       "being"
 [46] "have"       "has"        "had"        "having"     "do"
```

and this is a simple code to download stop words and removing them

.

```
import nltk
```

```
nltk.download('stopwords')
```

```
from nltk.corpus import stopwords
```

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



filtered_sentence = [w for w in wordDictA if not w in
stop_words]



print(filtered_sentence)
```

output :

```
['data', 'Science', 'job', 'sexiest', 'science', 'machine',
'learning', '21st', 'key', 'Data', 'century']
```

And now that we finished the TF section, we move onto the IDF part:

```
def computeIDF(docList):

   idfDict = {}

   N = len(docList)

```

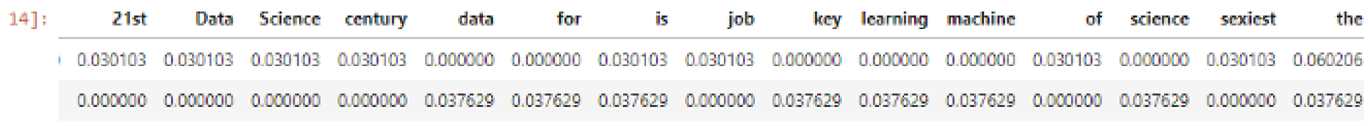
```
    idfDict = dict.fromkeys(docList[0].keys(), 0)

    for word, val in idfDict.items():

        idfDict[word] = math.log10(N / (float(val) + 1))



    return(idfDict)


#inputing our sentences in the log file


idfs = computeIDF([wordDictA, wordDictB])
```

and now we implement the idf formula , let’s finish with calculating the TFIDF

```
def computeTFIDF(tfBow, idfs):

    tfidf = {}
```

```
   for word, val in tfBow.items():

       tfidf[word] = val*idfs[word]

   return(tfidf)

#running our two sentences through the IDF:


idfFirst = computeTFIDF(tfFirst, idfs)


idfSecond = computeTFIDF(tfSecond, idfs)


#putting it in a dataframe


idf= pd.DataFrame([idfFirst, idfSecond])


print(idf)
```

output :



14]:

21st	Data	Science	century	data	for	is	job	key	learning	machine	of	science	sexiest	the
0.030103	0.030103	0.030103	0.030103	0.000000	0.000000	0.030103	0.030103	0.000000	0.000000	0.000000	0.030103	0.000000	0.030103	0.060206
0.000000	0.000000	0.000000	0.000000	0.037629	0.037629	0.037629	0.000000	0.037629	0.037629	0.037629	0.000000	0.037629	0.000000	0.037629

That was a lot of work. But it is handy to know, if you are asked to code TF-IDF from scratch in the future. However, this can be done a lot simpler thanks to sklearn library. Let's look at the example from them below:

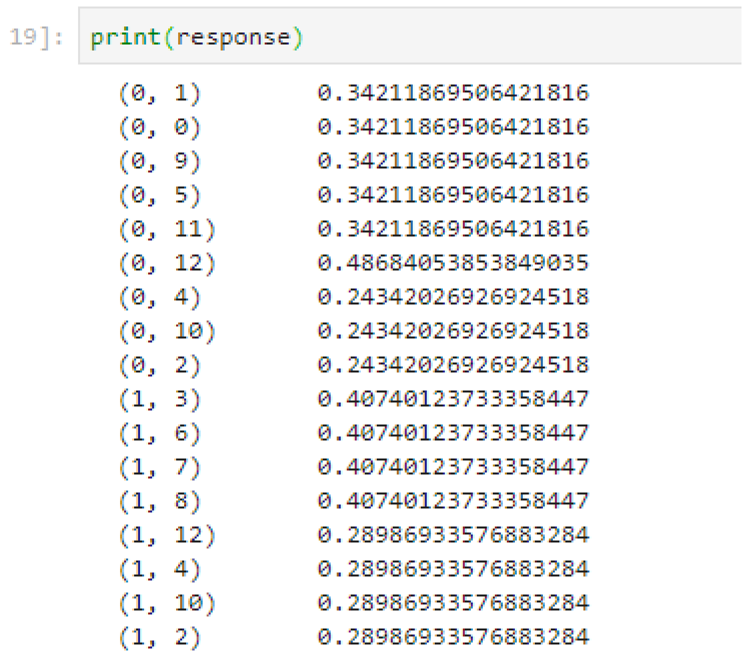
```
#first step is to import the library
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
#for the sentence, make sure all words are lowercase or you will
run #into error. for simplicity, I just made the same sentence
all #lowercase
```

```
firstV= "Data Science is the sexiest job of the 21st century"

secondV= "machine learning is the key for data science"

#calling the TfidfVectorizer

vectorize= TfidfVectorizer()

#fitting the model and passing our sentences right away:

response= vectorize.fit_transform([firstV, secondV])
```

and that’s the expected output :



```
19]: print(response)
```

```
  (0, 1)        0.34211869506421816
  (0, 0)        0.34211869506421816
  (0, 9)        0.34211869506421816
  (0, 5)        0.34211869506421816
  (0, 11)       0.34211869506421816
  (0, 12)       0.48684053853849035
  (0, 4)        0.24342026926924518
  (0, 10)       0.24342026926924518
  (0, 2)        0.24342026926924518
  (1, 3)        0.40740123733358447
  (1, 6)        0.40740123733358447
  (1, 7)        0.40740123733358447
  (1, 8)        0.40740123733358447
  (1, 12)       0.28986933576883284
  (1, 4)        0.28986933576883284
  (1, 10)       0.28986933576883284
  (1, 2)        0.28986933576883284
```