Assignment

What does tf-idf mean?

Tf-idf stands for *term frequency-inverse document frequency*, and the tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

One of the simplest ranking functions is computed by summing the tf-idf for each query term; many more sophisticated ranking functions are variants of this simple model.

Tf-idf can be successfully used for stop-words filtering in various subject fields including text summarization and classification.

</font>

How to Compute:

Typically, the tf-idf weight is composed by two terms: the first computes the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

- **TF:** Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization: $TF(t) = \frac{\text{Number of times term t appears in a document}}{\text{Total number of terms in the document}}$.
- **IDF:** Inverse Document Frequency, which measures how important a term is. 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 scale up the rare ones, by computing the following: $IDF(t) = \log_{e}\frac{\text{Total number of documents}}{\text{Number of documents with term t in it}}$. for numerical stabiltiy we will be changing this formula little bit $IDF(t) = \log_{e}\frac{\text{Total number of documents}}{\text{Number of documents with term t in it}+1}$.

Example

Consider a document containing 100 words wherein the word cat appears 3 times. The term frequency (i.e., tf) for cat is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word cat appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12. </p> </font>

Task-1

1. Build a TFIDF Vectorizer & compare its results with Sklearn:

- As a part of this task you will be implementing TFIDF vectorizer on a collection of text documents.

- You should compare the results of your own implementation of TFIDF vectorizer with that of sklearns implemenation TFIDF vectorizer.

- Sklearn does few more tweaks in the implementation of its version of TFIDF vectorizer, so to replicate the exact results you would need to add following things to your custom implementation of tfidf vectorizer:
  1. Sklearn has its vocabulary generated from idf sroted in alphabetical order
  2. Sklearn formula of idf is different from the standard textbook formula. Here the constant **"1"** is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions. $IDF(t) = 1 + \log_e \frac{1\ +\ \text{Total number of documents in collection}}{1+\text{Number of documents with term t in it}}$.
  3. Sklearn applies L2-normalization on its output matrix.
  4. The final output of sklearn tfidf vectorizer is a sparse matrix.

- Steps to approach this task:
  1. You would have to write both fit and transform methods for your custom implementation of tfidf vectorizer.
  2. Print out the alphabetically sorted voacb after you fit your data and check if its the same as that of the feature names from sklearn tfidf vectorizer.
  3. Print out the idf values from your implementation and check if its the same as that of sklearns tfidf vectorizer idf values.
  4. Once you get your voacb and idf values to be same as that of sklearns implementation of tfidf vectorizer, proceed to the below steps.
  5. Make sure the output of your implementation is a sparse matrix. Before generating the final output, you need to normalize your sparse matrix using L2 normalization. You can refer to this link https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html
  6. After completing the above steps, print the output of your custom implementation and compare it with sklearns implementation of tfidf vectorizer.
  7. To check the output of a single document in your collection of documents, you can convert the sparse matrix related only to that document into dense matrix and print it.

Note-1: All the necessary outputs of sklearns tfidf vectorizer have been provided as reference in this notebook, you can compare your outputs as mentioned in the above steps, with these outputs.
Note-2: The output of your custom implementation and that of sklearns implementation would match only with the collection of document strings provided to you as reference in this notebook. It would not match for strings that contain capital letters or punctuations, etc, because sklearn version of tfidf vectorizer deals with such strings in a different way. To know further details about how sklearn tfidf vectorizer works with such string, you can always refer to its official documentation.
Note-3: During this task, it would be helpful for you to debug the code you write with print statements wherever necessary. But when you are finally submitting the assignment, make sure your code is readable and try not to print things which are not part of this task.

Corpus

In [1]:

```
## SkLearn# Collection of string documents

corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]
```

SkLearn Implementation

In [14]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
print(vectorizer.fit(corpus))

skl_output = vectorizer.transform(corpus)
print(skl_output)
```

```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.float64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=True,
        stop_words=None, strip_accents=None, sublinear_tf=False,
        token_pattern='(?u)\\b\\w\\w+\\b', tokenizer=None, use_idf=True,
        vocabulary=None)
  (0, 8)	0.38408524091481483
  (0, 6)	0.38408524091481483
  (0, 3)	0.38408524091481483
  (0, 2)	0.5802858236844359
  (0, 1)	0.46979138557992045
  (1, 8)	0.281088674033753
  (1, 6)	0.281088674033753
  (1, 5)	0.5386476208856763
  (1, 3)	0.281088674033753
  (1, 1)	0.6876235979836938
  (2, 8)	0.267103787642168
  (2, 7)	0.511848512707169
  (2, 6)	0.267103787642168
  (2, 4)	0.511848512707169
  (2, 3)	0.267103787642168
  (2, 0)	0.511848512707169
  (3, 8)	0.38408524091481483
  (3, 6)	0.38408524091481483
  (3, 3)	0.38408524091481483
  (3, 2)	0.5802858236844359
  (3, 1)	0.46979138557992045
```

In [5]:

```
# sklearn feature names, they are sorted in alphabetic order by default.

print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

In [13]:

```
# Here we will print the sklearn tfidf vectorizer idf values after applying the fit met
hod
# After using the fit function on the corpus the vocab has 9 words in it, and each has
 its idf value.
print(vectorizer.idf_)
```

```
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
```

In [0]:

```
# shape of sklearn tfidf vectorizer output after applying transform method.

skl_output.shape
```

Out[0]:

```
(4, 9)
```

In [0]:

```
# sklearn tfidf values for first line of the above corpus.
# Here the output is a sparse matrix

print(skl_output[0])
```

```
  (0, 8)	0.38408524091481483
  (0, 6)	0.38408524091481483
  (0, 3)	0.38408524091481483
  (0, 2)	0.5802858236844359
  (0, 1)	0.46979138557992045
```

In [32]:

```
# sklearn tfidf values for first line of the above corpus.
# To understand the output better, here we are converting the sparse output matrix to d
ense matrix and printing it.
# Notice that this output is normalized using L2 normalization. sklearn does this by de
fault.

print(skl_output[3].toarray())
```

```
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Your custom implementation

In [54]:

```
# Write your code here.
# Make sure its well documented and readble with appropriate comments.
# Compare your results with the above sklearn tfidf vectorizer
# You are not supposed to use any other library apart from the ones given below
import warnings
warnings.filterwarnings("ignore")
from collections import Counter
from tqdm import tqdm
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy
import os


def wordscount(p):
    """
    This function returns count of words
    """
    count = 0
    words = p.split(" ")  #Spliting the document into words
    for ward in words:
        count +=1            #Calculate the count of words  in each document
    return count
def get_countdoc(corpus):
    """
    This function returns the no of words in each document
    """
    doc_count = []                       #initialize list as empty
    i =0
    for p in corpus:
        i +=1
        count = wordscount(p)       #Calling Count function
        temp = {'id_doc' : i, 'length_doc' : count}   #Stored as  dictionary for each document
        doc_count.append(temp)           #Append each document dictionary to list
    return doc_count

def get_worddoc(corpus):
    """
    This function returns each word count in each document"
    """
    i = 0
    freq = []            #Initialize list as empty
    for ivr in corpus :  #Iterating each document in corpus
        i += 1
        dict_freq = {}
        words = ivr.split(" ")  #Spliting document into words
        for wordi in words:
            wordi = wordi.lower()  #Converting all words to lower case
            if wordi in dict_freq:
                dict_freq[wordi] +=1  #Calculating each word count in each document
            else :
                dict_freq[wordi] = 1  #First time adding that word into dictionary is set to  1

            temp = {'id_doc' : i, 'freq_dit' : dict_freq}  #Adding each document with respective splited words count

        freq.append(temp)  #Appending dictinoary to final list
    return freq
```

```python
def calculateTF(docin,fredict):
    """
    This function returns TF Score for each word
    """
    tf_value = []         #Initilaize list as empty
    for values in fredict :   #Iterating every document in corpus
        id = values['id_doc']    #Find the document id of that corpus
        for k in values['freq_dit']: # Taking all words in each document
            temp = {'id_doc' : id, 'tf_score' : values['freq_dit'][k]/docin[id-1]['length_doc'], 'key' : k }  #Calcualate TF Score of words in each document
            tf_value.append(temp)  #Appending all documents TF Score to list
    return tf_value

def calculateidf(docin,fredict):
        """
        This function returns IDF Score for each word
        """
        idfscores = []
        t1 = []
        counter = 0
        for dict in fredict :  #Iterating every document in corpus
            counter +=1
            for k in dict['freq_dit'].keys():
                count = sum([k in tempDict['freq_dit'] for tempDict in fredict])  #No of documents containing that word
                temp = {'id_doc' : counter, 'IDFScore' :1.0 + (math.log((1+len(docin))/(1+count))), 'key': k} #Calulate the IDF Score of each word in document
                idfscores.append(temp) #Append each document IDF Score
        return idfscores

def calculatetfidf(tfscore,idfscore):
    """
    This function returns TF-IDF Score for each word
    """
    tfidfscore = []
    for j in idfscore:  #Iterating IDF Scores
        for i in tfscore: #Iterating TF Scores
            if j['key'] == i['key'] and j['id_doc'] == i['id_doc']:
                temp = {'id_doc' : j['id_doc'], 'tfidfscore' : j['IDFScore']*i['tf_score'], 'key' : i['key'] }  #Calculating TF-IDF Score for each word
        tfidfscore.append(temp)  #Appending each document TF-IDF Score
    return tfidfscore

def fit(corpus):
    """
    This function returns distinct features and their count
    """
    initial = set()
    if isinstance(corpus, (list,)):  #Check corpus is list or not
        for row in corpus :     #Iterating each document in corpus
            for w in row.split(" "):  #Spliting each document into words
                 initial.add(w)  #Appending distinct features to set

        finalwords = sorted(list(initial))  #Sorted the words
        print("Below are the Dimensional Features in sorted Order :")
        print(finalwords)  #Printing distinct features in corpus
        testin = {q : p for p,q in enumerate(finalwords)}   #Returns count of occurence of words
    return testin
```

```python
def transform(corpus,x,idfscore):
    """
    This function returns Sparse Matrix of corpus
    """
    rows = []
    columns = []
    values = []
    if isinstance(corpus, (list,)):  #Check corpus is list or not
        for i,j in enumerate(tqdm(corpus)):  #Iterating each document in corpus
            wordfree = dict(Counter(j.split()))  #Each word count in every document
            for ip in idfscore:  #Iterating each IDF Score
                if ip['id_doc']-1 == i:
                        p = ip['key']
                        q = ip['tfidfscore']
                        col_index = x.get(p, -1)  #Finding Column index for each word
                        if col_index !=-1 :
                            rows.append(i)   #Appending each row for each document
                            columns.append(col_index) #Appending each column index
                            values.append(q)  #Appending tfidfscore for each word
        return csr_matrix((values, (rows,columns)), shape=(len(corpus),len(x)))  #Returns the sparse Matrix
    else:
        print("you need to pass list of strings")


# Main Function
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]

docin = get_countdoc(corpus)   #By calling this function to get count of words in each document
fredict = get_worddoc(corpus) #By calling this function to get each word count in each document
tfscore = calculateTF(docin,fredict) #By calling this function  to calculate term frequency score of each word
idfscore = calculateidf(docin,fredict)#By calling this function to calculate Inverse Document frequency score of each document
tfidfscore = calculatetfidf(tfscore,idfscore)#By calling this function to calculate TF-IDF Score for each word
x = fit(corpus)  #By calling this function to get count of occurrence of distinct words
tfdprint = {i : j['IDFScore'] for i in x for j in idfscore if i==j['key'] } # IDF Score for distinct words
print("Below are the IDFScore Dimensional Features  :")
print(tfdprint)
p = transform(corpus,x,tfidfscore) #By calling this function to get matrix of corpus
finaltransform = normalize(p, axis=1, norm='l2') #Normalizing the Data
print("Final output for Sparse Matrix after normalization")
print(finaltransform)   #Print Sparse Matrix
print("Shape of Matrix is " + str(finaltransform.shape))
print("Here the output is a sparse matrix of finaltransorm[0] :")
print(finaltransform[0])
print("Here we are converting the sparse output matrix to dense matrix and printing it : ")
print(finaltransform[3].toarray())
print("Dense Matrix of output")
print(finaltransform.todense())
```

```
Below are the Dimensional Features in sorted Order :
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
Below are the IDFScore Dimensional Features  :
{'and': 1.916290731874155, 'document': 1.2231435513142097, 'first': 1.51082
56237659907, 'is': 1.0, 'one': 1.916290731874155, 'second': 1.9162907318741
55, 'the': 1.0, 'third': 1.916290731874155, 'this': 1.0}

100%|██████████████████████████████████████████████████████████████████████
████████████| 4/4 [00:00<00:00, 3986.03it/s]

Final output for Sparse Matrix after normalization
  (0, 1)	0.4697913855799205
  (0, 2)	0.580285823684436
  (0, 3)	0.3840852409148149
  (0, 6)	0.3840852409148149
  (0, 8)	0.3840852409148149
  (1, 1)	0.6876235979836937
  (1, 3)	0.2810886740337529
  (1, 5)	0.5386476208856762
  (1, 6)	0.2810886740337529
  (1, 8)	0.2810886740337529
  (2, 0)	0.511848512707169
  (2, 3)	0.267103787642168
  (2, 4)	0.511848512707169
  (2, 6)	0.267103787642168
  (2, 7)	0.511848512707169
  (2, 8)	0.267103787642168
  (3, 1)	0.4697913855799205
  (3, 2)	0.580285823684436
  (3, 3)	0.3840852409148149
  (3, 6)	0.3840852409148149
  (3, 8)	0.3840852409148149
Shape of Matrix is (4, 9)
Here the output is a sparse matrix of finaltransorm[0] :
  (0, 1)	0.4697913855799205
  (0, 2)	0.580285823684436
  (0, 3)	0.3840852409148149
  (0, 6)	0.3840852409148149
  (0, 8)	0.3840852409148149
Here we are converting the sparse output matrix to dense matrix and printin
g it :
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
Dense Matrix of output
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Task-2

2. Implement max features functionality:

- As a part of this task you have to modify your fit and transform functions so that your vocab will contain only 50 terms with top idf scores.

- This task is similar to your previous task, just that here your vocabulary is limited to only top 50 features names based on their idf values. Basically your output will have exactly 50 columns and the number of rows will depend on the number of documents you have in your corpus.

- Here you will be give a pickle file, with file name **cleaned_strings**. You would have to load the corpus from this file and use it as input to your tfidf vectorizer.

- Steps to approach this task:
  1. You would have to write both fit and transform methods for your custom implementation of tfidf vectorizer, just like in the previous task. Additionally, here you have to limit the number of features generated to 50 as described above.
  2. Now sort your vocab based in descending order of idf values and print out the words in the sorted voacb after you fit your data. Here you should be getting only 50 terms in your vocab. And make sure to print idf values for each term in your vocab.
  3. Make sure the output of your implementation is a sparse matrix. Before generating the final output, you need to normalize your sparse matrix using L2 normalization. You can refer to this link https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html
  4. Now check the output of a single document in your collection of documents, you can convert the sparse matrix related only to that document into dense matrix and print it. And this dense matrix should contain 1 row and 50 columns.

In [56]:

```
# Write your code here.
# Make sure its well documented and readble with appropriate comments.
# Compare your results with the above sklearn tfidf vectorizer
# You are not supposed to use any other library apart from the ones given below
import warnings
import pickle
warnings.filterwarnings("ignore")
from collections import Counter
from tqdm import tqdm
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy
import os


def wordscount(p):
    """
    This function returns count of words
    """
    count = 0
    words = p.split(" ")  #Spliting the document into words
    for ward in words:
        count +=1            #Calculate the count of words  in each document
    return count
def get_countdoc(corpus):
    """
    This function returns the no of words in each document
    """
    doc_count = []                          #initialize list as empty
    i =0
    for p in corpus:
        i +=1
        count = wordscount(p)       #Calling Count function
        temp = {'id_doc' : i, 'length_doc' : count}   #Stored as  dictionary for each d
ocument
        doc_count.append(temp)          #Append each document dictionary to list
    return doc_count

def get_worddoc(corpus):
    """
    This function returns each word count in each document"
    """
    i = 0
    freq = []             #Initialize list as empty
    for ivr in corpus :  #Iterating each document in corpus
        i += 1
        dict_freq = {}
        words = ivr.split(" ")  #Spliting document into words
        for wordi in words:
            wordi = wordi.lower()  #Converting all words to lower case
            if wordi in dict_freq:
                dict_freq[wordi] +=1  #Calculating each word count in each document
            else :
                dict_freq[wordi] = 1  #First time adding that word into dictionary is s
et to  1

            temp = {'id_doc' : i, 'freq_dit' : dict_freq}  #Adding each document with r
```

```
espective splited words count

        freq.append(temp)  #Appending dictinoary to final list
    return freq

def calculateTF(docin,fredict):
    """
    This function returns TF Score for each word
    """
    tf_value = []         #Initilaize list as empty
    for values in fredict :   #Iterating every document in corpus
        id = values['id_doc']    #Find the document id of that corpus
        for k in values['freq_dit']: # Taking all words in each document
            temp = {'id_doc' : id, 'tf_score' : values['freq_dit'][k]/docin[id-1]['leng
th_doc'], 'key' : k }  #Calcualate TF Score of words in each document
            tf_value.append(temp)  #Appending all documents TF Score to list
    return tf_value

def calculateidf(docin,fredict):
        """
        This function returns IDF Score for each word
        """
        idfscores = []
        t1 = []
        counter = 0
        for dict in fredict :  #Iterating every document in corpus
            counter +=1
            for k in dict['freq_dit'].keys():
                count = sum([k in tempDict['freq_dit'] for tempDict in fredict])  #No o
f documents containing that word
                temp = {'id_doc' : counter, 'IDFScore' :1.0 + (math.log((1+len(docin))/
(1+count))), 'key': k} #Calulate the IDF Score of each word in document
                idfscores.append(temp) #Append each document IDF Score
        return idfscores

def calculatetfidf(tfscore,idfscore):
    """
    This function returns TF-IDF Score for each word
    """
    tfidfscore = []
    for j in idfscore:  #Iterating IDF Scores
        for i in tfscore: #Iterating TF Scores
            if j['key'] == i['key'] and j['id_doc'] == i['id_doc']:
                temp = {'id_doc' : j['id_doc'], 'tfidfscore' : j['IDFScore']*i['tf_scor
e'], 'key' : i['key'] }  #Calculating TF-IDF Score for each word
        tfidfscore.append(temp)  #Appending each document TF-IDF Score
    return tfidfscore

def fit(corpus):
    """
    This function returns distinct features and their count
    """
    initial = set()
    if isinstance(corpus, (list,)):  #Check corpus is list or not
        for row in corpus :      #Iterating each document in corpus
            for w in row.split(" "):  #Spliting each document into words
                 initial.add(w)  #Appending distinct features to set

        finalwords = sorted(list(initial))  #Sorted the words
        testin = {q : p for p,q in enumerate(finalwords)}   #Returns count of occurence
of words
```

```
    return testin

def transform(corpus,x,idfscore):
    """
    This function returns Sparse Matrix of corpus
    """
    rows = []
    columns = []
    values = []
    if isinstance(corpus, (list,)):  #Check corpus is list or not
        for i,j in enumerate(tqdm(corpus)):  #Iterating each document in corpus
            wordfree = dict(Counter(j.split()))  #Each word count in every document
            for ip in idfscore:  #Iterating each IDF Score
                if ip['id_doc']-1 == i:
                        p = ip['key']
                        q = ip['tfidfscore']
                        col_index = x.get(p, -1)  #Finding Column index for each word
                        if col_index !=-1 :
                            rows.append(i)   #Appending each row for each document
                            columns.append(col_index) #Appending each column index
                            values.append(q)  #Appending tfidfscore for each word
        return csr_matrix((values, (rows,columns)), shape=(len(corpus),len(x)))  #Returns the sparse Matrix
    else:
        print("you need to pass list of strings")




 #Main Function
with open('cleaned_strings', 'rb') as f:
    corpus = pickle.load(f)


docin = get_countdoc(corpus)   #By calling this function to get count of words in each document
fredict = get_worddoc(corpus) #By calling this function to get each word count in each document
tfscore = calculateTF(docin,fredict) #By calling this function  to calculate term frequency score of each word
idfscore = calculateidf(docin,fredict)#By calling this function to calculate Inverse Document frequency score of each document
x = fit(corpus) #By calling this function to get count of occurrence of distinct words
tfdprint = {i: j['IDFScore'] for i in x for j in idfscore if i==j['key'] } # IDF Score for distinct words
feature = {ele[0]:ele[1] for ele in sorted(tfdprint.items() , reverse=True, key=lambda x: x[1])[:50]}
print("Below are the IDFScore Dimensional Features  :")
print(feature)
tfscore1 = [j for i in feature for j in tfscore if  i==j['key']]
idfscore1 = [j for i in feature for j in idfscore if  i==j['key']]
tfidfscore = calculatetfidf(tfscore,idfscore)#By calling this function to calculate TF-IDF Score for each word
x1= {key : value for i in feature for key,value in sorted(x.items()) if i==key}
x2 = [ele[0] for ele in sorted(x1.items() , reverse=True, key=lambda x: x[1])]
print("Below are the Dimensional Features  :")
print(x2)
x3 = {row : idx for idx, row in enumerate(x2)}
p = transform(corpus,x3,tfidfscore) #By calling this function to get matrix of corpus
finaltransform = normalize(p, axis=1, norm='l2') #Normalizing the Data
print("Final output for Sparse Matrix after normalization")
print(finaltransform) #Print Sparse Matrix
print("Shape of Matrix is " + str(finaltransform.shape))
print("Here the output is a sparse matrix of finaltransorm[0] :")
print(finaltransform[0])
print("Here we are converting the sparse output matrix to dense matrix and printing it
 : ")
print(finaltransform[3].toarray())
print("Dense Matrix of output")
print(finaltransform.todense())
```

```
Below are the IDFScore Dimensional Features  :
{'aailiyah': 6.922918004572872, 'abandoned': 6.922918004572872, 'abroad':
6.922918004572872, 'abstruse': 6.922918004572872, 'academy': 6.922918004572
872, 'accents': 6.922918004572872, 'accessible': 6.922918004572872, 'acclai
med': 6.922918004572872, 'accolades': 6.922918004572872, 'accurate': 6.9229
18004572872, 'accurately': 6.922918004572872, 'achille': 6.922918004572872,
'ackerman': 6.922918004572872, 'actions': 6.922918004572872, 'adams': 6.922
918004572872, 'add': 6.922918004572872, 'added': 6.922918004572872, 'admin
s': 6.922918004572872, 'admiration': 6.922918004572872, 'admitted': 6.92291
8004572872, 'adrift': 6.922918004572872, 'adventure': 6.922918004572872, 'a
esthetically': 6.922918004572872, 'affected': 6.922918004572872, 'affleck':
6.922918004572872, 'afternoon': 6.922918004572872, 'aged': 6.92291800457287
2, 'ages': 6.922918004572872, 'agree': 6.922918004572872, 'agreed': 6.92291
8004572872, 'aimless': 6.922918004572872, 'aired': 6.922918004572872, 'akas
ha': 6.922918004572872, 'akin': 6.922918004572872, 'alert': 6.9229180045728
72, 'alike': 6.922918004572872, 'allison': 6.922918004572872, 'allow': 6.92
2918004572872, 'allowing': 6.922918004572872, 'alongside': 6.92291800457287
2, 'amateurish': 6.922918004572872, 'amaze': 6.922918004572872, 'amazed':
6.922918004572872, 'amazingly': 6.922918004572872, 'amusing': 6.92291800457
2872, 'amust': 6.922918004572872, 'anatomist': 6.922918004572872, 'angel':
6.922918004572872, 'angela': 6.922918004572872, 'angelina': 6.9229180045728
72}
Below are the Dimensional Features  :
['angelina', 'angela', 'angel', 'anatomist', 'amust', 'amusing', 'amazingl
y', 'amazed', 'amaze', 'amateurish', 'alongside', 'allowing', 'allow', 'all
ison', 'alike', 'alert', 'akin', 'akasha', 'aired', 'aimless', 'agreed', 'a
gree', 'ages', 'aged', 'afternoon', 'affleck', 'affected', 'aesthetically',
'adventure', 'adrift', 'admitted', 'admiration', 'admins', 'added', 'add',
'adams', 'actions', 'ackerman', 'achille', 'accurately', 'accurate', 'accol
ades', 'acclaimed', 'accessible', 'accents', 'academy', 'abstruse', 'abroa
d', 'abandoned', 'aailiyah']

100%|███████████████████████████████████
████| 746/746 [00:00<00:00, 1233.85it/s]
```

```
Final output for Sparse Matrix after normalization
  (0, 19)	1.0
  (68, 25)	1.0
  (72, 20)	1.0
  (74, 18)	1.0
  (119, 16)	1.0
  (135, 8)	0.37796447300922725
  (135, 9)	0.37796447300922725
  (135, 13)	0.37796447300922725
  (135, 29)	0.37796447300922725
  (135, 31)	0.37796447300922725
  (135, 39)	0.37796447300922725
  (135, 46)	0.37796447300922725
  (176, 0)	1.0
  (181, 36)	1.0
  (192, 28)	1.0
  (193, 26)	1.0
  (216, 47)	1.0
  (222, 2)	1.0
  (225, 30)	1.0
  (227, 32)	1.0
  (241, 5)	1.0
  (270, 48)	1.0
  (290, 24)	1.0
  (333, 23)	1.0
  (334, 34)	1.0
  (341, 6)	1.0
  (344, 7)	1.0
  (348, 41)	1.0
  (377, 12)	1.0
  (409, 44)	1.0
  (430, 10)	1.0
  (457, 4)	1.0
  (461, 45)	1.0
  (465, 11)	1.0
  (475, 14)	1.0
  (493, 43)	1.0
  (500, 1)	1.0
  (548, 17)	0.7071067811865475
  (548, 49)	0.7071067811865475
  (608, 35)	1.0
  (612, 38)	1.0
  (620, 3)	1.0
  (632, 42)	1.0
  (644, 22)	0.7071067811865475
  (644, 37)	0.7071067811865475
  (664, 21)	1.0
  (667, 27)	1.0
  (691, 15)	1.0
  (697, 40)	1.0
  (722, 33)	1.0
Shape of Matrix is (746, 50)
Here the output is a sparse matrix of finaltransorm[0] :
  (0, 19)	1.0
Here we are converting the sparse output matrix to dense matrix and printin
g it :
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0.]]
Dense Matrix of output
[[0. 0. 0. ... 0. 0. 0.]
```

```
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

In [0]:

```
# Write your code here.
# Try not to hardcode any values.
# Make sure its well documented and readable with appropriate comments.
```

In [0]:

```

```