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.

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>

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: Sklearn has its vocabulary generated from idf sroted in alphabetical order 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 1 + Total number of documents in collection 1+Number of documents with term t in it

. IDF(t)=1+loge1 + Total number of documents in collection1+Number of documents with term t in it. Sklearn applies L2-normalization on its output matrix. The final output of sklearn tfidf vectorizer is a sparse matrix.

Steps to approach this task: You would have to write both fit and transform methods for your custom implementation of tfidf vectorizer. 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. Print out the idf values from your implementation and check if its the same as that of sklearns tfidf vectorizer idf values. Once you get your voacb and idf values to be same as that of sklearns implementation of tfidf vectorizer, proceed to the below steps. 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 After completing the above steps, print the output of your custom implementation and compare it with sklearns implementation of tfidf vectorizer. 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 [3]:
```
## 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 [179]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

In [180]:
```
# sklearn feature names, they are sorted in alphabetic order by defaul
t.

print(vectorizer.get_feature_names())
```

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

In [181]:
```
# Here we will print the sklearn tfidf vectorizer idf values after appl
ying the fit method
# After using the fit function on the corpus the vocab has 9 words in i
t, 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 [182]:
```
# shape of sklearn tfidf vectorizer output after applying transform met
hod.

skl_output.shape
```

Out[182]:
```
(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 [0]:
```
# sklearn tfidf values for first line of the above corpus.
# To understand the output better, here we are converting the sparse ou
tput matrix to dense matrix and printing it.
# Notice that this output is normalized using L2 normalization. sklearn
 does this by default.

print(skl_output[0].toarray())
```

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

custom implementation

In [1]:
```
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from tqdm import tqdm
import os
from tqdm import tqdm
```

```
import math
import numpy as np
import random
import string
import nltk
```

In [4]:
```
#reference links1: https://stackabuse.com/python-for-nlp-creating-bag-of-words-model-from-scratch/
#ref link2:Assignment 3 reference file and suggestions fromm AAIC Team
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]
```

creating method fit()

fit() method takes corpus(list if strings) as argument and returns a dict containing unique words as key and corrosponding freq as value

In [5]:
```
def fit(corpus):
    unique_words = set() # at first we will initialize an empty set
    # check if its list type or not
    if isinstance(corpus, (list,)):
        for row in corpus: # for each review in the dataset
            for word in row.split(" "): # for each word in the review.
 #split method converts a string into list of words
                if len(word) < 2:
                    continue
                unique_words.add(word)
        unique_words = sorted(list(unique_words))
        vocab = {j:i for i,j in enumerate(unique_words)}
        return vocab
    else:
        print("you need to pass list of sentance")
```

```
vocab=sorted(fit(corpus))
print(vocab)# this output matches the output given in this notebook for
 reference
```

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

creating method transform()

fit() method takes corpus(list if strings) as first argument and a dict containing unique words as key and corrosponding freq as value as second arguemts and retruns a sparse matrix as count vectors

In [7]:
```
from scipy.sparse import csr_matrix
from collections import Counter
def transform(corpus,vocab):
    rows = []
    columns = []
    values = []
    for idx,row in enumerate(corpus):
        #print("index is %d and value is %s" % (idx, val))
        #rint(idx,':',row)
        word_freq = dict(Counter(row.split()))
        #print(word_freq)#dict
        for word,freq in word_freq.items():
            #print(word,freq)
            col_idx=vocab.get(word,-1)
            if col_idx!=-1:
                rows.append(idx)
                columns.append(col_idx)
                values.append(freq)
    return csr_matrix((values, (rows,columns)), shape=(len(corpus),len(
vocab)))


vocab=fit(corpus)
```

```
trans=transform(corpus,vocab)
print(trans.toarray())
```

```
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

creating method idf()

idf() methpd takes corpus and list of unique words as input and returns a dict containing each unique words as key and corresponding idf as value

In [9]:
```
def idf(corpus,vocab):
    N=len(corpus)
    #print(N)
    idf_values={}#initializing dict

    for word in tqdm(vocab):#iterating through each word present in lis
t of vocab
        count=0
        for row in corpus:#iterating through each row of the corpus
            if word in row.split():#if unique word present,incrementing
 the count
                count+=1
        #print(num documents containing word)
        idf_values[word] =1+np.log((1+N)/(1 + count)) #calculating idf

    return idf_values


IDF_Values=idf(corpus,vocab)
print(IDF_Values)
```

```
100%|██████████| 9/9 [00:00<00:00, 11763.40it/s]
```

```
{'and': 1.916290731874155, 'document': 1.2231435513142097, 'first': 1.5
```

```
108256237659907, 'is': 1.0, 'one': 1.916290731874155, 'second': 1.91629

0731874155, 'the': 1.0, 'third': 1.916290731874155, 'this': 1.0}
```

creating method tf()

tf() methpd takes corpus and list of unique words as input and returns a dict containing each unique words as key and a list of corresponding tf in each line as value

In [10]:
```
def tf(corpus,vocab):
    tf_values = {}#initializing dict

    for word in tqdm(vocab):#iterating through each word present in lis
t of unique words
        tf_vector = []
        for row in corpus:#iterating through each row of the corpus
            doc_freq = 0
            for sem_token in row.split():#checking words in each senten
ce in the corpus matches the words in the list of unique words
                if word == sem_token:#if same incremant the count
                    doc_freq += 1
            #print(doc_freq)
            word_tf = doc_freq/len(row.split())#calculating tf
            tf_vector.append(word_tf)

        #print(sent_tf_vector)
        tf_values[word] = tf_vector

    return tf_values


#printing dict containing each unique present in given corpus as key an
d its corresponding tf vector as values
TF_Values=tf(corpus,vocab)
print(TF_Values)
```

```
100%|
```

```
| 9/9 [00:00<?, ?it/s]
```

```
{'and': [0.0, 0.0, 0.16666666666666666, 0.0], 'document': [0.2, 0.33333
33333333333, 0.0, 0.2], 'first': [0.2, 0.0, 0.0, 0.2], 'is': [0.2, 0.16
666666666666666, 0.16666666666666666, 0.2], 'one': [0.0, 0.0, 0.1666666
6666666666, 0.0], 'second': [0.0, 0.16666666666666666, 0.0, 0.0], 'th
e': [0.2, 0.16666666666666666, 0.16666666666666666, 0.2], 'third': [0.
0, 0.0, 0.16666666666666666, 0.0], 'this': [0.2, 0.16666666666666666,
0.16666666666666666, 0.2]}
```

creating method tfidf()

- tfidf() methpd takes two arguments.Dict containing each words from the list of unique words for each line is passed as the first argument whereas dict containing each unique word and its corrosponding IDF value is passed as second argument.
- we get the result as 'l2' normalized sparse matrix.

In [11]:
```
from sklearn.preprocessing import normalize
from scipy.sparse import csr_matrix

def tfidf(TF_Values,IDF_Values):
    tfidf_values = []
    for token in tqdm(TF_Values.keys()):
        tfidf_sentences = []
        for tf_sentence in TF_Values[token]:
            tf_idf_score = tf_sentence * IDF_Values[token] #getting tfi
df value of each word for each line
            tfidf_sentences.append(tf_idf_score)
        tfidf_values.append(tfidf_sentences)

        tf_idf_model = np.transpose(tfidf_values)

    sp_mat=csr_matrix(tf_idf_model)
    norm_sp_mat=normalize(sp_mat, norm='l2', axis=1)

    return norm_sp_mat
```

```
tfidf_out=tfidf(TF_Values,IDF_Values)
print(tfidf_out[0])
```

```
100%|██████████| 9/9 [00:00<00:00, 4684.63it/s]
```

```
  (0, 1)        0.4697913855799205
  (0, 2)        0.580285823684436
  (0, 3)        0.3840852409148149
  (0, 6)        0.3840852409148149
  (0, 8)        0.3840852409148149
```

In [12]:
```
#printing shape of the resultant output
print(tfidf_out.shape)
```

```
(4, 9)
```

In [14]:
```
#calling all the custom build fun sequentially for collective output
unique_words=sorted(fit(corpus))#returns list of unique words
IDF_Values=idf(corpus,vocab)#returns dic containg idf val of each word
TF_Values=tf(corpus,vocab)#returns tf val of each word in each sentence
output=tfidf(TF_Values,IDF_Values)#returns req ifidf val in aparse matr
ix
print(output[0])
```

```
100%|██████████| 9/9 [00:00<00:00, 8642.11it/s]
100%|██████████| 9/9 [00:00<00:00, 9009.24it/s]
100%|██████████| 9/9 [00:00<00:00, 9024.32it/s]
```

```
  (0, 1)        0.4697913855799205
  (0, 2)        0.580285823684436
  (0, 3)        0.3840852409148149
  (0, 6)        0.3840852409148149
  (0, 8)        0.3840852409148149
```

In [15]:
```
print("converting sparse matrix into dense matrix: ")
#converting sparse matrix into dense matrix
print(output[0].toarray())
```

```
converting sparse matrix into dense matrix:
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Conclusion :

All outputs obtained by custom implementation are exactly matching with outputs given in this notebook for reference.

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 [16]:
```
# Below is the code to load the cleaned_strings pickle file provided
# Here corpus is of list type

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

# printing the length of the corpus loaded
print("Number of documents in corpus = ",len(corpus))
```

```
Number of documents in corpus =  746
```

In [17]:
```
print(corpus[0:5]) #printing 5 sentences for ref
```

```
['slow moving aimless movie distressed drifting young man', 'not sure l
ost flat characters audience nearly half walked', 'attempting artiness
black white clever camera angles movie disappointed became even ridicul
ous acting poor plot lines almost non existent', 'little music anything
speak', 'best scene movie gerardo trying find song keeps running head']
```

creating top_50_idf_words()

This method that takes corpus as input and returns tfidf values in sparse matrix

In [18]:
```
def top_50_idf_words(corpus):
    #applying fit method to get dict containg unique words and its fre
q,later converting keys of dict to find idf value
    temp=fit(corpus)
    vocab_list=list(temp.keys())

    # idf values of each unique words
    idf_all_words=idf(corpus,vocab_list)

    #sorting list in decending order
    sorted_idf_list=sorted(idf_all_words.items(),key=lambda x:x[1],reve
rse=True)

    #print(sorted_idf_list)
    return dict(sorted_idf_list[:50])


top_50_idf_vocab=top_50_idf_words(corpus)
print(top_50_idf_vocab)
```

```
100%|██████████| 2886/2886 [00:04<00:00, 679.44it/s]
```

```
{'aailiyah': 6.922918004572872, 'abandoned': 6.922918004572872, 'abroa
d': 6.922918004572872, 'abstruse': 6.922918004572872, 'academy': 6.9229
18004572872, 'accents': 6.922918004572872, 'accessible': 6.922918004572
872, 'acclaimed': 6.922918004572872, 'accolades': 6.922918004572872, 'a
ccurate': 6.922918004572872, 'accurately': 6.922918004572872, 'achill
e': 6.922918004572872, 'ackerman': 6.922918004572872, 'actions': 6.9229
18004572872, 'adams': 6.922918004572872, 'add': 6.922918004572872, 'add
ed': 6.922918004572872, 'admins': 6.922918004572872, 'admiration': 6.92
2918004572872, 'admitted': 6.922918004572872, 'adrift': 6.9229180045728
72, 'adventure': 6.922918004572872, 'aesthetically': 6.922918004572872,
'affected': 6.922918004572872, 'affleck': 6.922918004572872, 'afternoo
n': 6.922918004572872, 'aged': 6.922918004572872, 'ages': 6.92291800457
2872, 'agree': 6.922918004572872, 'agreed': 6.922918004572872, 'aimles
s': 6.922918004572872, 'aired': 6.922918004572872, 'akasha': 6.92291800
4572872, 'akin': 6.922918004572872, 'alert': 6.922918004572872, 'alik
```

```
e': 6.922918004572872, 'allison': 6.922918004572872, 'allow': 6.9229180
04572872, 'allowing': 6.922918004572872, 'alongside': 6.92291800457287
2, 'amateurish': 6.922918004572872, 'amaze': 6.922918004572872, 'amaze
d': 6.922918004572872, 'amazingly': 6.922918004572872, 'amusing': 6.922
918004572872, 'amust': 6.922918004572872, 'anatomist': 6.92291800457287
2, 'angel': 6.922918004572872, 'angela': 6.922918004572872, 'angelina':
6.922918004572872}
```

In [19]:
```
#checking if we have 50 item in dict or not
print(len(top_50_idf_vocab))
```

```
50
```

In [20]:
```
#using transform to get count vectors
vec_top_50=transform(corpus,top_50_idf_vocab)
print(vec_top_50.toarray())
```

```
[[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 [21]:
```
#printing choosen words
top_50_idf_list=list(top_50_idf_vocab)
print(top_50_idf_list)
```

```
['aailiyah', 'abandoned', 'abroad', 'abstruse', 'academy', 'accents',
'accessible', 'acclaimed', 'accolades', 'accurate', 'accurately', 'achi
lle', 'ackerman', 'actions', 'adams', 'add', 'added', 'admins', 'admira
tion', 'admitted', 'adrift', 'adventure', 'aesthetically', 'affected',
'affleck', 'afternoon', 'aged', 'ages', 'agree', 'agreed', 'aimless',
'aired', 'akasha', 'akin', 'alert', 'alike', 'allison', 'allow', 'allow
ing', 'alongside', 'amateurish', 'amaze', 'amazed', 'amazingly', 'amusi
ng', 'amust', 'anatomist', 'angel', 'angela', 'angelina']
```

In [22]:
```
T_val_=tf(corpus,top_50_idf_list)
#print(T_val_)

TF_idf_value=tfidf(T_val_,top_50_idf_vocab)
print(TF_idf_value[0])
```

```
100%|██████████| 50/50 [00:00<00:00, 486.73it/s]
100%|██████████| 50/50 [00:00<00:00, 416.82it/s]
```

```
  (0, 30)       1.0
```

In [23]:
```
print(TF_idf_value[0].toarray())
```

```
[[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. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.
  0. 0.]]
```

In [ ]:
```

```