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.

Task-1

Corpus

In [178]:
```
## 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 default.

print(vectorizer.get_feature_names())
```

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

In [181]:
```
# Here we will print the sklearn tfidf vectorizer idf values after applying t
# After using the fit function on the corpus the vocab has 9 words in it, and

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 method.

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 output m
# Notice that this output is normalized using L2 normalization. sklearn does

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

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

MY 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
nltk.download('punkt')
```

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

Out[1]:
```
True
```

In [2]:
```
#ref: https://stackabuse.com/python-for-nlp-creating-bag-of-words-model-from-
#ref: https://stackabuse.com/python-for-nlp-creating-tf-idf-model-from-scratc
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 [3]:

```
def fit(corpus):
    word_freq={}

    for sentence in tqdm(corpus): #iterating through each sentence one by one
        words=sentence.split()  #each sentence is converted into individual w
        #print(tokens)
        '''iterating through each word of the list and checking if that word
           If present we increment the value of the corresponding word(key) b
           if not present we povide value of the corresponding word(key)=1'''

        for word in words:
            if word not in word_freq.keys():
                word_freq[word]=1
            else:
                word_freq[word]+=1

    return word_freq


'''printing dict containing unique words and corrosponding freq
word_freq=fit(corpus)
print(word_freq)

output: {'this': 4, 'is': 4, 'the': 4, 'first': 2, 'document': 4, 'second': 1

#printing list of alphabetically ordered unique words present in the corpus
unique_words=sorted(fit(corpus))
print(unique_words)# this output matches the output given in this notebook fo
```

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

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

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 [5]:
```python
from scipy.sparse import csr_matrix
from collections import Counter
def transform(corpus,bow_dict):
    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=bow_dict.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(bow_di


Bow_dict=fit(corpus)
trans_vec=transform(corpus,Bow_dict)
print(trans_vec.toarray())

```

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

[[0 0 1 0 4 0 0 0 0]
 [0 1 0 0 5 0 0 0 0]
 [0 3 0 0 3 0 0 0 0]
 [0 0 1 0 4 0 0 0 0]]
```

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 [6]:
```python
def idf(corpus,unique_words):
    N=1+len(corpus)
    #print(N)
    word_idf_values={}#initializing dict

    for word in tqdm(unique_words):#iterating through each word present in Li
        num_doc_containing_word=0
        for sentence in corpus:#iterating through each sentence of the corpus
            if word in sentence.split():#if unique word present,incrementing
                num_doc_containing_word+=1
        #print(num_doc_containing_word)
        word_idf_values[word] =1+np.log(N/(1 + num_doc_containing_word)) #cal

    return word_idf_values

#printing dict containing each unique word present in given corpus as key and
IDF_Values=idf(corpus,unique_words)
print(IDF_Values)# this output matches the output given in this notebook for
```

```
100%|██████████████████████████████████████████████████████████████████████████
█████████| 9/9 [00:00<00:00, 458.50it/s]

{'and': 1.916290731874155, 'document': 1.2231435513142097, 'first': 1.510825623
7659907, 'is': 1.0, 'one': 1.916290731874155, 'second': 1.916290731874155, 'th
e': 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 [7]:
```python
def tf(corpus,unique_words):
    tf_values = {}#initializing dict

    for word in tqdm(unique_words):#iterating through each word present in li
        sent_tf_vector = []
        for sentence in corpus:#iterating through each sentence of the corpus
            doc_freq = 0
            for sem_token in sentence.split():#checking words in each sentenc
                if word == sem_token:#if same incremant the count
                    doc_freq += 1
            #print(doc_freq)
            word_tf = doc_freq/len(sentence.split())#calculating tf
            sent_tf_vector.append(word_tf)

        #print(sent_tf_vector)
        tf_values[word] = sent_tf_vector

    return tf_values


#printing dict containing each unique present in given corpus as key and its
TF_Values=tf(corpus,unique_words)
print(TF_Values)
```

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

{'and': [0.0, 0.0, 0.16666666666666666, 0.0], 'document': [0.2, 0.3333333333333
333, 0.0, 0.2], 'first': [0.2, 0.0, 0.0, 0.2], 'is': [0.2, 0.16666666666666666,
0.16666666666666666, 0.2], 'one': [0.0, 0.0, 0.16666666666666666, 0.0], 'secon
d': [0.0, 0.16666666666666666, 0.0, 0.0], 'the': [0.2, 0.16666666666666666, 0.1
6666666666666666, 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 [12]:

```
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 tfidf val
            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<?, ?it/s]

  (0, 0)	0.3840852409148149
  (0, 1)	0.3840852409148149
  (0, 2)	0.3840852409148149
  (0, 3)	0.580285823684436
  (0, 4)	0.4697913855799205
```

In [13]:

```
#printing shape of the resultant output
print(tfidf_out.shape)
```

```
(4, 9)
```

In [15]:
```python
#calling all the custom build fun sequentially for collective output
unique_words=sorted(fit(corpus))#returns list of unique words
IDF_Values=idf(corpus,unique_words)#returns dic containg idf val of each wor
TF_Values=tf(corpus,unique_words)#returns tf val of each word in each sentenc
cus_output=tfidf(TF_Values,IDF_Values)#returns req ifidf val in aparse matrix
print(cus_output[0])
```

```
100%|██████████████████████████████████████████████████████████████████
█████████████████| 4/4 [00:00<?, ?it/s]
100%|██████████████████████████████████████████████████████████████████
█████████████████| 9/9 [00:00<?, ?it/s]
100%|██████████████████████████████████████████████████████████████████
█████████████████| 9/9 [00:00<?, ?it/s]
100%|██████████████████████████████████████████████████████████████████
█████████████████| 9/9 [00:00<?, ?it/s]

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

In [16]:
```python
print("converting sparse matrix into dense matrix: ")
#converting sparse matrix into dense matrix
print(cus_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 same as 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 (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 [17]:
```
# 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 [18]:
```
print(corpus[0:5]) #printing 5 sentences for ref
```

```
['slow moving aimless movie distressed drifting young man', 'not sure lost flat
characters audience nearly half walked', 'attempting artiness black white cleve
r camera angles movie disappointed became even ridiculous acting poor plot line
s almost non existent', 'little music anything speak', 'best scene movie gerard
o trying find song keeps running head']
```

In [19]:
```
#printing dict containing unique words from the corpus as key and its corrosp
#vocab_dic=fit(corpus)
#print(vocab_dic.keys())
```

In [20]:
```
#vocab_list=sorted(fit(corpus))
#print(vocab_list)
```

creating top_50_idf_words()

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

In [21]:
```
def top_50_idf_words(corpus):
    #applying fit method to get dict containg unique words and its freq,later
    temp=fit(corpus)
    vocab_list=list(temp.keys())

    #getting 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],reverse=Tr

    #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%|████████████████████████████████████████████████████████████████
███| 746/746 [00:00<00:00, 62184.76it/s]
100%|████████████████████████████████████████████████████████████████
██| 2897/2897 [00:02<00:00, 1011.85it/s]

{'aimless': 6.922918004572872, 'distressed': 6.922918004572872, 'drifting': 6.9
22918004572872, 'nearly': 6.922918004572872, 'attempting': 6.922918004572872,
'artiness': 6.922918004572872, 'existent': 6.922918004572872, 'gerardo': 6.9229
18004572872, 'emptiness': 6.922918004572872, 'effort': 6.922918004572872, 'mess
ages': 6.922918004572872, 'buffet': 6.922918004572872, 'science': 6.92291800457
2872, 'teacher': 6.922918004572872, 'baby': 6.922918004572872, 'owls': 6.922918
004572872, 'florida': 6.922918004572872, 'muppets': 6.922918004572872, 'perso
n': 6.922918004572872, 'overdue': 6.922918004572872, 'screenplay': 6.9229180045
72872, 'post': 6.922918004572872, 'practically': 6.922918004572872, 'structur
e': 6.922918004572872, 'tightly': 6.922918004572872, 'constructed': 6.922918004
572872, 'vitally': 6.922918004572872, 'occurs': 6.922918004572872, 'content':
6.922918004572872, 'fill': 6.922918004572872, 'dozen': 6.922918004572872, 'high
est': 6.922918004572872, 'superlative': 6.922918004572872, 'require': 6.9229180
04572872, 'puzzle': 6.922918004572872, 'solving': 6.922918004572872, 'fit': 6.9
22918004572872, 'pulls': 6.922918004572872, 'punches': 6.922918004572872, 'grap
hics': 6.922918004572872, 'number': 6.922918004572872, 'th': 6.922918004572872,
'insane': 6.922918004572872, 'massive': 6.922918004572872, 'unlockable': 6.9229
18004572872, 'properly': 6.922918004572872, 'aye': 6.922918004572872, 'rocks':
6.922918004572872, 'doomed': 6.922918004572872, 'conception': 6.92291800457287
2}
```

In [22]:
```
#checking if we have 50 item in dict or not
print(len(top_50_idf_vocab))
```

```
50
```

In [23]:
```
#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 [25]:
```
#printing choosen words
top_50_idf_list=list(top_50_idf_vocab)
print(top_50_idf_list)
```

```
['aimless', 'distressed', 'drifting', 'nearly', 'attempting', 'artiness', 'exis
tent', 'gerardo', 'emptiness', 'effort', 'messages', 'buffet', 'science', 'teac
her', 'baby', 'owls', 'florida', 'muppets', 'person', 'overdue', 'screenplay',
'post', 'practically', 'structure', 'tightly', 'constructed', 'vitally', 'occur
s', 'content', 'fill', 'dozen', 'highest', 'superlative', 'require', 'puzzle',
'solving', 'fit', 'pulls', 'punches', 'graphics', 'number', 'th', 'insane', 'ma
ssive', 'unlockable', 'properly', 'aye', 'rocks', 'doomed', 'conception']
```

In [28]:
```
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, 279.86it/s]
100%|██████████████████████████████████████████████████
███████| 50/50 [00:00<00:00, 432.19it/s]

  (0, 0)        0.5773502691896257
  (0, 1)        0.5773502691896257
  (0, 2)        0.5773502691896257
```

In [27]:
```
print(TF_idf_value[0].toarray())
```

```
[[0.57735027 0.57735027 0.57735027 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.        ]]
```

In [ ]:
```

```