# 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

corpus1 = [
     '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 [5]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus1)
skl_output = vectorizer.transform(corpus1)
```

Out[5]:
```
<4x9 sparse matrix of type '<class 'numpy.float64'>'
        with 21 stored elements in Compressed Sparse Row format>
```

In [6]:
```
# 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 [7]:
```
# 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 [8]:
```
# shape of sklearn tfidf vectorizer output after applying transform met
hod.

skl_output.shape
```

Out[8]:
```
(4, 9)
```

In [9]:
```
# 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 [10]:
```
# 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]]
```

## Your custom implementation

In [11]:
```
# 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 giv
en below

from collections import Counter
from tqdm import tqdm
from scipy.sparse import csr_matrix
import math
import re
import string
from functools import reduce
from math import log
import pandas as pd
import operator
from sklearn.preprocessing import normalize
import numpy
```

In [12]:
```
def fit(corpus1):
    '''This function return vocab and the idf'''
    unique_words = set()#In ths set we will store the word so that we g
et unique word
    for row in corpus1:#This for loop will visit each row and split tha
```

```
t row and make union with unique_word
        unique_words=unique_words.union(row.split())
    unique_words=list(unique_words)#Here we are converting set to a lis
t so that we can sort it easly
    unique_words.sort()#sorting the list
    vocab = {j:i for i,j in enumerate(unique_words)}#Here we are storin
g word and column in a dictonary

    td=len(corpus1)#Here we are storing the total no of document in the
 corpus
    td=td+1#we added 1 according to the formula of scikit-learn
    b=[]#In this list we will store idf of each word
    c=0 #We will keep count in c of the document n which word appear fr
om vocab
    for i in list(vocab):#This for loop will itterate each word in voca
b for idf
        c=1#Here we stated from 1 to according to formula used in sciki
t-learn
        for row in corpus1:#This for loop will visit through each docum
ent in corpus to check presence of word
            if i in row.split():#Here we split the document on space an
d used the membership function to check presence of word in document
                c=c+1#We increse c by 1 when we find the word in docume
nt
        idf=1+math.log(td/c)#Here we use scikit-learn formula to calcul
ate idf
        b.append(idf)#Storing idf in b
    df_vocab={i:j for i,j in zip(list(vocab),b)}#Here we are storing wo
rd and idf in a dictonary

    return vocab,df_vocab#returning vocab and idf
```

In [14]:
```
vocab,df_vocab=fit(corpus1)#Calling fit function on corpus
#vocab
df_vocab
```

Out[14]:
```
{'and': 1.916290731874155,
 'document': 1.2231435513142097,
 'first': 1.5108256237659907,
```

```
 'is': 1.0,
 'one': 1.916290731874155,
 'second': 1.916290731874155,
 'the': 1.0,
 'third': 1.916290731874155,
 'this': 1.0}
```

In [15]:
```
print(list(vocab))#Converted dict to list and printed the corpus
```

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

In [16]:
```
print(list(df_vocab.values()))#Printing the values of dict df_vocab whi
ch is idf
```

```
[1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0, 1.9162
90731874155, 1.916290731874155, 1.0, 1.916290731874155, 1.0]
```

In [17]:
```
def transform(corpus1,vocab):
    '''This function return tfidf'''
    rows=[]#Here we will store row no of non zero values
    columns=[]#Here we will store column no of non zero values
    values=[]#Here we will store non zero values
    #All the three rows column and values contain information about sam
e element in diff list
    for i,row in enumerate(tqdm(corpus1)):#This will go through each do
cument in the corpus
        l=len(row.split())#n l we are storing the no of token in documn
t
        word_freq=dict(Counter(row.split()))#It return count of each to
ken present in document
        for word, freq in word_freq.items():#In this for loop we will c
alculate tfidf and store row ,column and values in respective list
            rows.append(i)
            columns.append(vocab.get(word))
            t=freq/l
            t=t*df_vocab.get(word)
            values.append(t)
    tfidf=csr_matrix((values, (rows,columns)), shape=(len(corpus1),len(
```

```
vocab)))#Here we are covering list rows,columns,values to sparce matrix
    tfidf=normalize(tfidf)#Here we used l2 normalization according to s
cikit-learn
    return tfidf
```

In [18]:
```
tfidf=transform(corpus1,vocab)#Calling transform function
```

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

In [19]:
```
print(tfidf[0])#Printing tfidf of first document in corpus
```

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

In [20]:
```
print(skl_output[0].toarray())
#print(skl_output[0])
```

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

# observation

Hence the result is matched with sklearn implementation

# 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 [24]:
```
# 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:
    corpus2 = pickle.load(f)

# printing the length of the corpus loaded
print("Number of documents in corpus = ",len(corpus2))
```

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

In [26]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus2)
skl_output = vectorizer.transform(corpus2)
skl_output
```

Out[26]:
```
<746x2886 sparse matrix of type '<class 'numpy.float64'>'
        with 6964 stored elements in Compressed Sparse Row format>
```

In [27]:
```
# sklearn feature names, they are sorted in alphabetic order by defaul
t.
#first 50 feature names
print(vectorizer.get_feature_names()[:50])
```

```
['aailiyah', 'abandoned', 'ability', 'abroad', 'absolutely', 'abstrus
e', 'abysmal', 'academy', 'accents', 'accessible', 'acclaimed', 'accola
des', 'accurate', 'accurately', 'accused', 'achievement', 'achille', 'a
ckerman', 'act', 'acted', 'acting', 'action', 'actions', 'actor', 'acto
rs', 'actress', 'actresses', 'actually', 'adams', 'adaptation', 'add',
'added', 'addition', 'admins', 'admiration', 'admitted', 'adorable', 'a
drift', 'adventure', 'advise', 'aerial', 'aesthetically', 'affected',
'affleck', 'afraid', 'africa', 'afternoon', 'age', 'aged', 'ages']
```

In [28]:
```
# sklearn feature names, they are sorted in alphabetic order by defaul
t.
#last 50 feature names
print(vectorizer.get_feature_names()[-50:])
```

```
['wonder', 'wondered', 'wonderful', 'wonderfully', 'wong', 'wont', 'wo
o', 'wooden', 'word', 'words', 'work', 'worked', 'working', 'works', 'w
orld', 'worry', 'worse', 'worst', 'worth', 'worthless', 'worthwhile',
'worthy', 'would', 'wouldnt', 'woven', 'wow', 'wrap', 'write', 'write
```

```
r', 'writers', 'writing', 'written', 'wrong', 'wrote', 'yardley', 'yaw
n', 'yeah', 'year', 'years', 'yelps', 'yes', 'yet', 'young', 'younger',
'youthful', 'youtube', 'yun', 'zillion', 'zombie', 'zombiez']
```

In [29]:
```
print(vectorizer.idf_)
```

```
[6.922918   6.922918   6.22977082 ... 6.922918   6.5174529  6.922918  ]
```

In [30]:
```
skl_output.shape
```

Out[30]:
```
(746, 2886)
```

In [31]:
```
# sklearn tfidf values for first line of the above corpus.
# Here the output is a sparse matrix

print(skl_output[0])
```

```
  (0, 2878)     0.35781145622317734
  (0, 2287)     0.3377679916467555
  (0, 1653)     0.35781145622317734
  (0, 1651)     0.16192317905848022
  (0, 1545)     0.30566026894803877
  (0, 720)      0.4123943870778812
  (0, 688)      0.4123943870778812
  (0, 53)       0.4123943870778812
```

In [32]:
```
# 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. 0. ... 0. 0. 0.]]
```

In [33]:
```
#custom implementation
```

```
#tf_compute
def fit(corpus2):
    '''This function return vocab and the idf'''
    unique_words = set()#In ths set we will store the word so that we g
et unique word
    for row in corpus2:#This for loop will visit each row and split tha
t row and make union with unique_word
        unique_words=unique_words.union(row.split())
    unique_words=list(unique_words)#Here we are converting set to a lis
t so that we can sort it easly
    unique_words.sort()#sorting the list
    vocab = {j:i for i,j in enumerate(unique_words)}#Here we are storin
g word and column in a dictonary

    td=len(corpus2)#Here we are storing the total no of document in the
 corpus
    td=td+1#we added 1 according to the formula of scikit-learn
    b=[]#In this list we will store idf of each word
    c=0 #We will keep count in c of the document n which word appear fr
om vocab
    for i in list(vocab):#This for loop will itterate each word in voca
b for idf
        c=1#Here we stated from 1 to according to formula used in sciki
t-learn
        for row in corpus2:#This for loop will visit through each docum
ent in corpus to check presence of word
            if i in row.split():#Here we split the document on space an
d used the membership function to check presence of word in document
                c=c+1#We increse c by 1 when we find the word in docume
nt
        idf=1+math.log(td/c)#Here we use scikit-learn formula to calcul
ate idf
        b.append(idf)#Storing idf in b
    df_vocab={i:j for i,j in zip(list(vocab),b)}#Here we are storing wo
rd and idf in a dictonary

    return vocab,df_vocab#returning vocab and idf
```

In [34]:
```
vocab,df_vocab=fit(corpus2)#Calling fit function on corpus
```

In [44]:
```
print(list(vocab))#Converted dict to list and printed the corpus
```

In [43]:
```
print(list(df_vocab.values()))#Printing the values of dict df_vocab whi
ch is idf
```

In [85]:
```
def transform(corpus2,vocab):
    '''This function return tfidf'''
    rows=[]#Here we will store row no of non zero values
    columns=[]#Here we will store column no of non zero values
    values=[]#Here we will store non zero values
    #All the three rows column and values contain information about sam
e element in diff list
    for i,row in enumerate(tqdm(corpus2)):#This will go through each do
cument in the corpus
        l=len(row.split())#n l we are storing the no of token in documn
t
        word_freq=dict(Counter(row.split()))#It return count of each to
ken present in document
        for word, freq in word_freq.items():#In this for loop we will c
alculate tfidf and store row ,column and values in respective list
            rows.append(i)
            columns.append(vocab.get(word))
            t=freq/l
            t=t*df_vocab.get(word)
            values.append(t)
    tfidf=csr_matrix((values, (rows,columns)), shape=(len(corpus2),len(
vocab)))#Here we are covering list rows,columns,values to sparce matrix
    tfidf=normalize(tfidf)#Here we used l2 normalization according to s
cikit-learn
    return tfidf
```

In [37]:
```
tfidf=transform(corpus2,vocab)#Calling transform function
```

```
100%|██████████| 746/746 [00:00<00:00, 19144.10it/s]
```

In [38]:
```
print(tfidf[0])#Printing tfidf of first document in corpus
```

```
  (0, 53)       0.4123943870778812
  (0, 689)      0.4123943870778812
  (0, 721)      0.4123943870778812
  (0, 1549)     0.30566026894803877
  (0, 1655)     0.16192317905848022
  (0, 1657)     0.35781145622317734
  (0, 2294)     0.3377679916467555
  (0, 2888)     0.35781145622317734
```

In [39]:
```
print(skl_output[0].toarray())
```

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