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.

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:
    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 [ ]:
```
## 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 [ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

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

skl_output.shape
```

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

In [ ]:
```
# 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 [ ]:
```
# 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 [1]:
```
# 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 operator
from sklearn.preprocessing import normalize
import numpy
```

In [41]:
```
# define a fit function():

def fit(dataset):

    if isinstance(dataset,(list,)): # checks whether input argument is
 a list of sentence
        unique_words = set()        # define a set to store unique word
s

        for sent in tqdm(dataset):
            for word in sent.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)   # add unique words from the co
```

```
rpus into the set

        unique_words = sorted(list(unique_words)) # sort the words alph
abetically

        print("(A) The required features names in alphabetical order :
\n",unique_words,'\n')

        idf_words = []              # create a list to store idf values
        for word in unique_words:
            count = 0
            for sent in dataset:
                mylist = set(sent.split(" "))
                for j in mylist:
                    if word == j:
                        count+=1
                    else:
                        continue

            denominator = 1 + count         # denominator of idf formul
a
            numerator = 1 + len(dataset)    # numerator of idf formula

            idf_word = (1 + math.log(numerator/denominator)) # formula
 to compute idf
            idf_words.append(idf_word)                # append values
 to the list

        print("(B) The desired idf values for the unique words :\n",idf
_words)

        # create dictionary with index number for each unique word
        vocab = {j:i for i,j in enumerate(unique_words)}

        # combine words and idf values into a dict{}
        dictionary = dict(zip(unique_words,idf_words))

        return vocab,idf_words,dictionary      # return the required en
tities
```

```
    else:
        print("You Need to pass list of sentences/documents")


# Collection of documents provided for the task:
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]

# call the function:
vocab,idf_words,dictionary = fit(corpus)
```

```
100%|████████████████████████████████████████████████████████████| 4/4 [00:00<00:00, 4007.94it/s]
```

```
(A) The required features names in alphabetical order :
 ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 't
his']

(B) The desired idf values for the unique words :
 [1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0, 1.916
290731874155, 1.916290731874155, 1.0, 1.916290731874155, 1.0]
```

In [43]:
```
# define a transform():

def transform(dataset,vocab,dictionary):
    rows = []                           # list for storing row number
    columns = []                        # list for storing column number
    values = []                         # list for storing tfidf values

    if isinstance(dataset,(list,)):     # checks whether input argument
 is a list of sentence

        for idx,row in tqdm(enumerate(dataset)):
            word_freq = dict(Counter(row.split())) #count frequency of
```

```
 each word & make dict()

            for word,freq in word_freq.items():  # for each unique word
 in the review.
                if len(word) < 2:
                    continue

                col_idx = vocab.get(word,-1) #get the value for the key
s(words) provided

                if col_idx != -1:

                    rows.append(idx)         # row number

                    columns.append(col_idx)   # column number

                    tf_word = freq/(len(row.split())) # compute term fr
equency

                    values.append(tf_word*dictionary[word]) # compute t
fidf values

        return normalize(csr_matrix((values,(rows,columns)),shape=(len(
dataset),len(vocab))))
    else:
        print("You need to pass list of documents")


# Call the function with required parameters:

myoutput_task1 = transform(corpus,vocab,dictionary) # store returned ou
tput into other variable

# print statements:

print("(A) My output shape is: ",myoutput_task1.shape,"\n")

print("(B) The Tfidf values for first document in sparse format:\n")
```

```
print(myoutput_task1[0],"\n")

print("(C) The dense representation of output matrix:\n")

print(myoutput_task1[0].toarray())
```

```
4it [00:00, 4012.73it/s]

(A) My output shape is:  (4, 9)

(B) The Tfidf values for first document in sparse format:

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

(C) The dense representation of output matrix:

[[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 [46]:
```
# 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_task2 = pickle.load(f)

# printing the length of the corpus loaded
print("Number of documents in corpus = ",len(corpus_task2))
```

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

In [146]:
```
# define a fit()

def fit(dataset):
```

```
    if isinstance(dataset,(list,)):  # checks whether input argument is
 a list of sentence
        unique_words = set()         # define a set to store unique wor
ds
        for sent in tqdm(dataset):
            for word in sent.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)  # add unique words from the cor
pus into the set

        unique_words = sorted(list(unique_words)) #sort list of unique
 words alphabetically

        idf_words = []                   # list to store idf values
        for word in unique_words:
            count = 0
            for sent in dataset:
                mylist = set(sent.split(" "))
                for j in mylist:
                    if word == j:
                        count+=1
                    else:
                        continue

            denominator = 1 + count          # denominator of idf formu
la
            numerator = 1 + len(dataset)     # numerator of idf formula

            idf_word = (1 + math.log(numerator/denominator)) # compute
 idf using formula
            idf_words.append(idf_word)

        vocab = {j:i for i,j in enumerate(unique_words)} #create dictio
nary with index number for each unique word

        x = dict(zip(unique_words,idf_words)) # make a dict() by combin
ing words & idf values
```

```
        # sort the dict() based on idf values
        dictionary = {k: v for k, v in sorted(x.items(), key=lambda ite
m: item[1],reverse = True)}

        # extract top features based on idf values
        dictionary = dict(list(dictionary.items())[0:50])

        vocab_top50 = []        # list for storing top 50 feature words
 based on idf value
        for key in dictionary:
            for word in vocab:
                if key == word:
                    vocab_top50.append(word)

        return vocab_top50,idf_words,dictionary # return required entit
ies
    else:
        print("You Need to pass list of sentences/documents")

# call the function with desired arguments:

vocab_top50,idf_words,dictionary = fit(corpus_task2)

# print statements:

print("(A) No of feature names generated :\n")
print(len(vocab_top50))
print("\n")

print("(B) Top features based on idf values: \n")
print(vocab_top50)
print("\n")

print("(C) Idf values for Top 50 terms : \n")
print(dictionary)
```

```
100%|██████████| 746/746 [00:00<00:00, 124051.49it/s]
```

```
(A) No of feature names generated :
```

```
(A) No of feature names generated :

50


(B) Top features based on idf values:

['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']


(C) Idf values for Top 50 terms :

{'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 [148]:

```
# define a transform():

def transform(dataset,vocab,dictionary):

    rows = []                           # list for storing row number
    columns = []                        # list for storing column number
    values = []                         # list for storing tfidf values

    if isinstance(dataset,(list,)): # checks whether input argument is
 a list of sentence

        for idx,row in tqdm(enumerate(dataset)):
            word_freq = dict(Counter(row.split())) #count frequency of
 each word & make dict

            for word,freq in word_freq.items():  # for each unique word
 in the review.
                if len(word) < 2 :
                    continue
                if word in vocab:
                    col_idx = vocab.index(word)  # get the index of tha
t particular word

                    if col_idx != -1:

                        rows.append(idx)         # row number

                        columns.append(col_idx)  # column number

                        tf_word = freq/(len(row.split())) # compute ter
m frequency

                        values.append(tf_word*dictionary[word]) # add t
fidf values to list

        # return the required output
        return normalize(csr_matrix((values,(rows,columns)),shape=(len(
```

```
dataset),len(vocab))))
    else:
        print("You need to pass list of documents")

#call the function:

myoutput_task2 = transform(corpus_task2,vocab_top50,dictionary)

# print statements:

print("(A) The class of the output matrix  :\n")
print(type(myoutput_task2))
print("\n")

print("(B) The shape of the output matrix :\n")
print(myoutput_task2.shape)
print("\n")

print("(C) The shape of the output matrix for single document :\n")
print(myoutput_task2[0].shape)
print("\n")

print("(D) The sparse representation for single document :\n")
print(myoutput_task2[0])
print("\n")

print("(E) The dense matrix representation for single document :\n")
print(myoutput_task2[0].toarray())
```

```
746it [00:00, 29922.07it/s]
```

```
(A) The class of the output matrix  :

<class 'scipy.sparse.csr.csr_matrix'>


(B) The shape of the output matrix :

(746, 50)
```

```
(C) The shape of the output matrix for single document :

(1, 50)


(D) The sparse representation for single document :

  (0, 30)       1.0


(E) The dense matrix representation for single document :

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