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. $IDF(t) = 1+\log_{e}\frac{1\text{ }+\text{ Total number of documents in collection}} {1+\text{Number of documents with term t in it}}.$
  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 [2]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

In [3]:
```
# 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 [4]:
```
# Here we will print the sklearn tfidf vectorizer idf values after applying the fit method
# 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 [5]:
```
# shape of sklearn tfidf vectorizer output after applying transform method.

skl_output.shape
```

Out[5]:
```
(4, 9)
```

In [6]:
```
# 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 [7]:

```
# sklearn tfidf values for first line of the above corpus.
# To understand the output better, here we are converting the sparse output matrix to dense matrix and printing i
# 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 [8]:

```
# 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

from collections import Counter
# from tqdm import tqdm
from tqdm.notebook import tqdm
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy
```

In [9]:

```
def fit(data_set):
    '''
    Input
    -----
    data_set : Data coupus - Lists of sentences

    Output
    ------
    unique_words : List of all unique words from the data corpus
    idf_word_values : The Inverse Document Frequency values corresponding to each word
    '''

    unique_words_set = set() #For storing all unique words in data_set

#     Reference : Assignment_3_Reference : https://colab.research.google.com/drive/1Y_K1iQV_wv7Z7I63axwMQJp1XJzgc

    for rows in data_set:
        for word in rows.split():
            if len(word) < 2: #Screening words
                continue
            unique_words_set.add(word) #Adding words with length >=2.
#     print(unique_words_set)

    unique_words = sorted(list(unique_words_set)) #Sorting the words aplhabetical order
#     print(unique_words)
    vocabulary = {j:i for i,j in enumerate(unique_words)}
#     print(vocabulary)

    no_of_total_documents = len(data_set)

    # Calculating IDF value

    idf_word_values = {}   # Dict for storing idf values
    for word in unique_words:
        count = 0
        for words in data_set: # Main loop starts here : Iterating over data_set

            if word in words.split(): # Checking for the presence of unique_words 'word'
                count += 1
            idf_word_values[word] = 1 + math.log((no_of_total_documents+1) / (count+1)) # IDF calcualtion
                                # Adding 1 : Reference : Task 1 reference

    return vocabulary, idf_word_values
```

In [10]:

```
vocabulary, idf_word_value = fit(corpus)
```

Results after `fit()` function

In [11]:

```python
print('Alphabetically sorted vocabulary :')
print('Sk-learn implementation: ', vectorizer.get_feature_names())
print('Custom implementation  : ', list(vocabulary.keys()))
print('\nBoth implementation results are same : ',
      vectorizer.get_feature_names() == list(vocabulary.keys()), '\n')

print('-'*100)
print('-'*100)

print('\nIDF values :')
print('Sk-learn implementation: ', vectorizer.idf_)
print('Custom implementation  : ', list(idf_word_value.values()))
print('\nBoth implementation results are same : ',
      [float(value) for value in vectorizer.idf_] == list(idf_word_value.values()))
```

```
Alphabetically sorted vocabulary :
Sk-learn implementation:  ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
Custom implementation  :  ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']

Both implementation results are same :  True

----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------

IDF values :
Sk-learn implementation:  [1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
Custom implementation  :  [1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0, 1.916290731874155, 1.9
16290731874155, 1.0, 1.916290731874155, 1.0]

Both implementation results are same :  True
```

In [12]:

```python
def transform(data_set, vocabulary_, idf_word_values):
    '''
    Input
    -----
    data_set : Data coupus - Lists of sentences
    vocabulary_ : Dictionary contaion unique words extracted from data_set
    idf_word_values : The Inverse Document Frequency values corresponding to each word

    Output
    ------
    normalized_sparse_matrix : Normalized Sparse Matrix of shape
                               (len(data_set),len(unique_words)) and L2 normalized
    '''

#     Reference : Assignment_3_Reference
    mat_rows = []
    mat_columns = []
    mat_values = []

#     vocab = {j:i for i,j in enumerate(unique_words)}

    for index, rows in enumerate(tqdm(data_set)):
        row_word_freq = dict(Counter(rows.split())) #Creating word_freq counter for TF calculation
        for word in rows.split():
            if len(word) < 2: #Screening words
                continue

            if word in vocabulary_.keys():
                tf_value = row_word_freq.get(word) / len(rows.split()) #calculating TF values

#                 print(tf_value, word, row_word_freq[word])

                tf_idf_value = tf_value * idf_word_values.get(word)  # TF-IDF value = TF * IDF
#                 print(round(tf_value,3),'\t',word,'\t\t',round(tf_idf_value,3))

#                 Reference : Assignment_3_Reference
                col_index = vocabulary_.get(word, -1)

                if col_index != -1:
                    mat_rows.append(index)
                    mat_columns.append(col_index)
                    mat_values.append(tf_idf_value)
#                 print(word ,vocabulary_[word], col_index)

#     Creating Sparse Matrix, with shape = (len(data_set),len(unique_words))
    sparse_matrix = csr_matrix((mat_values, (mat_rows,mat_columns)),
                               shape = (len(data_set),len(vocabulary_)))

#     Normalizing the sparse matrix using 'l2' normalization
#     https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html
    normalized_sparse_matrix = normalize(sparse_matrix, norm ='l2')

    return normalized_sparse_matrix
```

In [13]:

```python
sparse_matrix = transform(corpus, vocabulary, idf_word_value)
```

Results after `transform()` function

In [14]:
```python
# Shape of sparse matrix

print('Sk-learn implementation : ', skl_output.shape)
print('Custom implementation   : ', sparse_matrix.shape)
```

```
Sk-learn implementation :  (4, 9)
Custom implementation   :  (4, 9)
```

In [15]:
```python
# TF-IDF values for first line of the above corpus : Output after L2 normalization

print('Sk-learn implementation')
print('-'*23)
print(skl_output[0], '\n')

print('Custom implementation')
print('-'*21)
print(sparse_matrix[0])
```

```
Sk-learn implementation
-----------------------
  (0, 8)	0.38408524091481483
  (0, 6)	0.38408524091481483
  (0, 3)	0.38408524091481483
  (0, 2)	0.5802858236844359
  (0, 1)	0.46979138557992045

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

In [16]:
```python
# TF-IDF values for first line of the above corpus : Output after L2 normalization

print('Sk-learn implementation')
print('-'*23)
print(skl_output[0].toarray(), '\n'*2)

print('Custom implementation')
print('-'*21)
print(sparse_matrix[0].toarray())
```

```
Sk-learn implementation
-----------------------
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]


Custom implementation
---------------------
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Values of Custom implementation are same as Sk-learn 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 [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]:
```
def fit(data_set, top_features):
    '''
    Input
    -----
    data_set : Data coupus - Lists of sentences

    Output
    ------
    unique_words : List of all unique words from the data corpus
    idf_word_values : The Inverse Document Frequency values corresponding to each word
    '''

    unique_words_set = set() #For storing all unique words in data_set

#     Reference : Assignment_3_Reference : https://colab.research.google.com/drive/1Y_K1iQV_wv7Z7I63axwMQJp1XJzgc

    for rows in data_set:
        for word in rows.split():
            if len(word) < 2: #Screening words
                continue
            unique_words_set.add(word) #Adding words with length >=2.
#     print(unique_words_set)

    unique_words = sorted(list(unique_words_set)) #Sorting the words aplhabetical order
#     print(unique_words)

    no_of_total_documents = len(data_set)

    # Calculating IDF value

    idf_word_values = {}   # Dict for storing idf values
    for word in unique_words:
        count = 0
        for words in data_set: # Main loop starts here : Iterating over data_set

            if word in words.split(): # Checking for the presence of unique_words 'word'
                count += 1
            idf_word_values[word] = 1 + math.log((no_of_total_documents+1) / (count+1)) # IDF calcualtion
                            # Adding 1 : Reference : Task 1 reference

    '''
    https://stackoverflow.com/a/613218
    https://docs.python.org/3/howto/sorting.html#ascending-and-descending

    x = {'a': 2, 'b': 4, 'd': 3, 'h': 1, 'c': 0}
    {k: v for k, v in sorted(x.items(), key=lambda item: item[1], reverse=True)}
    {'b': 4, 'd': 3, 'a': 2, 'h': 1, 'c': 0}
    '''

#     Sorting IDF values in descending order
    sorted_idf_word_values = {}
    count = 0
    for k, v in sorted(idf_word_values.items(),key=lambda item: item[1], reverse=True):
        if top_features == count:
            break
        sorted_idf_word_values[k] = v
        count += 1
```

```python
#     print('sorted_idf_word_values length : ', len(sorted_idf_word_values))

#     Creating vocabulary only to contain sorted IDF words
    vocabulary = {}
    index = 0
    for words in unique_words:
        if words in sorted_idf_word_values.keys():
            vocabulary[words] = index
            index += 1

    return vocabulary, sorted_idf_word_values
```

In [19]:

```python
TOP_FEATURES = 50

vocabulary , sorted_idf_word_value = fit(corpus, TOP_FEATURES)
```

In [20]:

```python
# printing word and IDF values for each term in the vocabulary

for index, word in enumerate(vocabulary):
    if len(word) < 4:
        print(f'{index+1}. {word}\t\t\t{sorted_idf_word_value[word]}')
    elif len(word) < 8:
        print(f'{index+1}. {word}\t\t{sorted_idf_word_value[word]}')
    elif len(word) > 12:
        print(f'{index+1}. {word}\t{sorted_idf_word_value[word]}')
    else:
        print(f'{index+1}. {word}\t\t{sorted_idf_word_value[word]}')
```

```
1. aailiyah             6.922918004572872
2. abandoned            6.922918004572872
3. abroad               6.922918004572872
4. abstruse             6.922918004572872
5. academy              6.922918004572872
6. accents              6.922918004572872
7. accessible           6.922918004572872
8. acclaimed            6.922918004572872
9. accolades            6.922918004572872
10. accurate            6.922918004572872
11. accurately          6.922918004572872
12. achille             6.922918004572872
13. ackerman            6.922918004572872
14. actions             6.922918004572872
15. adams               6.922918004572872
16. add                 6.922918004572872
17. added               6.922918004572872
18. admins              6.922918004572872
19. admiration          6.922918004572872
20. admitted            6.922918004572872
21. adrift              6.922918004572872
22. adventure           6.922918004572872
23. aesthetically       6.922918004572872
24. affected            6.922918004572872
25. affleck             6.922918004572872
26. afternoon           6.922918004572872
27. aged                6.922918004572872
28. ages                6.922918004572872
29. agree               6.922918004572872
30. agreed              6.922918004572872
31. aimless             6.922918004572872
32. aired               6.922918004572872
33. akasha              6.922918004572872
34. akin                6.922918004572872
35. alert               6.922918004572872
36. alike               6.922918004572872
37. allison             6.922918004572872
38. allow               6.922918004572872
39. allowing            6.922918004572872
40. alongside           6.922918004572872
41. amateurish          6.922918004572872
42. amaze               6.922918004572872
43. amazed              6.922918004572872
44. amazingly           6.922918004572872
45. amusing             6.922918004572872
46. amust               6.922918004572872
47. anatomist           6.922918004572872
48. angel               6.922918004572872
49. angela              6.922918004572872
50. angelina            6.922918004572872
```

In [21]:

```python
# Utilizing the previously defined `transform()` function

sparse_matrix = transform(corpus, vocabulary, sorted_idf_word_value)
```

In [22]:

```
print('Shape of Sparse Matrix is :',sparse_matrix.shape)
```

```
Shape of Sparse Matrix is : (746, 50)
```

Step 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 [23]:

```
#https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.todense.html#scipy.sparse.csc_matri
print('Dense matrix : ',sparse_matrix[0].todense())
print('\nShape of dense matrix :',sparse_matrix[0].todense().shape)
```

```
Dense matrix :  [[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. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0.]]

Shape of dense matrix : (1, 50)
```