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 [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)
import numpy as np
```

In [3]:
```
# sklearn get_feature_name, they get sorted unique value

print(vectorizer.get_feature_names())
```

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

In [4]:
```python
# 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]:
```python
# after apply transform method we will see the shape of sklearn tfidf.

skl_output.shape
```

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

In [6]:
```python
# 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]:
```python
# 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 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]]
```

In [8]:
```
# we can print all of corpus

print(skl_output.toarray())
```

```
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Your custom implementation

In [9]:
```
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 [10]:
```
def sort_dic_ByValue(test_dict):

    # here we sort the value in decreasing order

    x = sorted(test_dict.items(), key = lambda kv:(kv[1], kv[0]),reverse=True)[0:50]
    x = dict(x)
    print(x)
```

In [11]:
```
test_dict = {'a':2,'b':50,'c':4,'d':60, 'e':10}
r=sort_dic_ByValue(test_dict)
```

```
{'d': 60, 'b': 50, 'e': 10, 'c': 4, 'a': 2}
```

In [12]:
```
def word_contain_reviews(dataset,word):
    freq_of_word = 0
    for review in dataset:
        if word in review:
            freq_of_word +=1
    return freq_of_word
```

In [13]:
```
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]
freq_of_word = word_contain_reviews(corpus,'document')
print(freq_of_word)
```

```
3
```

In [14]:
```
# fit function for tf-idf vectorizer
# with the help of this function we will find the unique_features and their idf values
def fit(dataset):
    unique_words=set() #make empty set
    idf_value=[]

    if isinstance(dataset, (list,)): # check our dataset list our not.
        for row in dataset:

            for word in (row.split()):
                if len(word) < 2:
                    continue # skip those words which length less than 2
                unique_words.add(word)
        total_num_of_rev=len(dataset) # here we find total number of reviews
        for word in unique_words:
            tot_rev_contain_word=word_contain_reviews(dataset,word) #here we finding particular word contain review
            # apply logarithm to find idf value
            idf_value.append(1+math.log((1+total_num_of_rev)/(1+tot_rev_contain_word)))
        unique_words=list(unique_words)
        features=zip(unique_words,idf_value) # merge unique words and idf values
        sorted_features=dict(sorted(features, key = lambda kv:(kv[0], kv[1]))[0:50])
        #sort the feature key wise and change in dic
        return sorted_features
    else:
        print('you need to pass a list')
```

In [15]:
```
sorted_features=fit(corpus)
```

```
In [16]: sorted_features
```

```
Out[16]: {'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 [17]: print('custom features')
         print(list(sorted_features.keys()))
         print('sklearn features')
         print(vectorizer.get_feature_names())
```

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

```
In [18]: # import numpy
         # because sklearn's idf_ is in numpy.ndarray format
         import numpy as np

         # print idf of the gained features from the given corpus
         print('custom idfs:')
         print(np.array(list(sorted_features.values())))

         #sklearn idfs
         print('sklearn idfs:')
         print(vectorizer.idf_)
```

```
custom idfs:
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
sklearn idfs:
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
```

In [19]:
```
def transform(dataset, vocab):
    rows = []              # row values for sparse matrix
    columns = []            # columns values for sparse matrix
    values = []             # frequency of word


    if  isinstance(dataset, (list,)):

        for idx, row in enumerate(tqdm(dataset)):

            word_freq = dict(Counter(row.split()))

            for word in (row.split()):
                if len(word)<2:
                    continue
                idf = vocab.get(word)              #find the column index from vocab
                tf = word_freq[word]/len(row)      #calculate tf value
                tfidf = tf*float(idf)

                col_index = list(vocab.keys()).index(word)
                rows.append(idx)                               #storing the row number
                columns.append(col_index)                      #store the  column number
                values.append(tfidf)                          #storing the values

        # here we perform the L2 normalization
        tfidf_matrix = csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab)))
        L2_normalized_matrix = normalize(tfidf_matrix)
        return L2_normalized_matrix
    else:
        print("you need to pass the dataset in list format")
```

In [20]:
```
custom_output = transform(corpus,sorted_features)
print(custom_output[0])
# print('shape:',tfidf.toarray().shape)
```

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

  (0, 1)	0.46979138557992045
  (0, 2)	0.5802858236844359
  (0, 3)	0.3840852409148149
  (0, 6)	0.3840852409148149
  (0, 8)	0.3840852409148149
```

In [21]:
```
# print sparse matrix in matrix form
print('custom output \n',custom_output[0].toarray())
```

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

In [22]:
```
# shape of idf array
print('shape of idf array')
print(custom_output.toarray().shape)
```

```
shape of idf array
(4, 9)
```

Task-2

implement max feature functionality

In [23]:
```python
# 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 [24]:
```python
def fit(dataset):
    unique_words=set() #make empty set
    idf_value=[]

    if isinstance(dataset, (list,)): # check our dataset list our not.
        for row in dataset:

            for word in (row.split()):
                if len(word) < 2:
                    continue # skip those words which length less than 2
                unique_words.add(word)
        total_num_of_rev=len(dataset) # here we find total number of reviews
        for word in unique_words:
            tot_rev_contain_word=word_contain_reviews(dataset,word) #here we finding particular word contain review
            # apply logarithm to find idf value
            idf_value.append(1+math.log((1+total_num_of_rev)/(1+tot_rev_contain_word)))
        unique_words=list(unique_words)
        features=zip(unique_words,idf_value) # merge unique words and idf values
        task2_sorted=sorted(features, key = lambda kv:(kv[0], kv[1]))[-50:]
        task2_sorted_features=dict(task2_sorted)
        #sort the feature key wise and change in dic
        return task2_sorted_features
    else:
        print('you need to pass a list')
```

In [41]:
```
def transform(dataset, vocab):
    rows = []              # row values for sparse matrix
    columns = []            # columns values for sparse matrix
    values = []             # frequency of word


    if  isinstance(dataset, (list,)):

        for idx, row in enumerate(tqdm(dataset)):

            word_freq = dict(Counter(row.split()))

            for word in (row.split()):
                if len(word)<2:
                    continue

                if word not in list(vocab.keys()):
                    continue

                else:
                    idf = vocab.get(word)                #find the column index from vocab
                    tf = word_freq[word]/len(row)        #calculate tf value
                    tfidf = tf*idf

                    col_index = list(vocab.keys()).index(word)
                    rows.append(idx)                            #storing the row number
                    columns.append(col_index)                   #store the  column number
                    values.append(tfidf)                       #storing the values

        # here we perform the L2 normalization
        tfidf_matrix = csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab)))
        L2_normalized_matrix = normalize(tfidf_matrix)
        return L2_normalized_matrix
    else:
        print("you need to pass the dataset in list format")
```

In [42]:
```
task2_sorted_features = fit(corpus)
```

In [43]:
```
# sklearn implementation
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=50)
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

In [44]:
```
# custom features
print('custom top 50 features')
print(list(task2_sorted_features.keys()))

# sklearn features
print('sklearn top 50 features')
print(vectorizer.get_feature_names())
```

```
custom top 50 features
['wonder', 'wondered', 'wonderful', 'wonderfully', 'wong', 'wont', 'woo', 'wooden', 'word', 'words', 'work', 'worked',
'working', 'works', 'world', 'worry', 'worse', 'worst', 'worth', 'worthless', 'worthwhile', 'worthy', 'would', 'wouldn
t', 'woven', 'wow', 'wrap', 'write', 'writer', 'writers', 'writing', 'written', 'wrong', 'wrote', 'yardley', 'yawn',
'yeah', 'year', 'years', 'yelps', 'yes', 'yet', 'young', 'younger', 'youthful', 'youtube', 'yun', 'zillion', 'zombie',
'zombiez']
sklearn top 50 features
['acting', 'actors', 'also', 'bad', 'best', 'better', 'cast', 'character', 'characters', 'could', 'even', 'ever', 'eve
ry', 'excellent', 'film', 'films', 'funny', 'good', 'great', 'like', 'little', 'look', 'love', 'made', 'make', 'movi
e', 'movies', 'much', 'never', 'no', 'not', 'one', 'plot', 'real', 'really', 'scenes', 'script', 'see', 'seen', 'sho
w', 'story', 'think', 'time', 'watch', 'watching', 'way', 'well', 'wonderful', 'work', 'would']
```

In [45]:
```
vectorizer.idf_
```

Out[45]:
```
array([3.97847903, 4.67162621, 4.39718936, 3.62708114, 4.57154275,
       4.78285184, 4.67162621, 4.57154275, 4.15032928, 4.39718936,
       4.03254625, 4.48057097, 4.84347646, 4.97700786, 2.7718781 ,
       4.67162621, 4.78285184, 3.78742379, 4.18207798, 4.00514727,
       4.72569343, 4.62033291, 4.57154275, 4.48057097, 4.67162621,
       2.71822539, 4.48057097, 4.72569343, 4.72569343, 4.35796865,
       2.89756631, 3.57301392, 4.35796865, 4.57154275, 4.08970466,
       4.78285184, 4.67162621, 4.03254625, 4.48057097, 4.78285184,
       4.57154275, 4.67162621, 3.95250354, 4.72569343, 4.67162621,
       4.52502273, 4.11955762, 4.67162621, 4.67162621, 4.28386067])
```

In [46]:
```
print('custom idfs:\n',np.array(list(task2_sorted_features.values())))
```

```
custom idfs:
 [4.43801135 6.922918   4.52502273 6.22977082 6.922918   6.922918
 5.82430572 6.5174529  5.31348009 6.00662727 4.32022832 6.922918
 6.5174529  5.82430572 5.82430572 6.922918   5.21816991 5.21816991
 4.62033291 6.922918   6.922918   6.22977082 4.24876936 6.922918
 6.922918   6.922918   6.922918   5.53662364 5.53662364 6.922918
 4.97700786 5.53662364 6.22977082 6.922918   6.922918   6.922918
 6.5174529  4.84347646 5.05111583 6.922918   5.53662364 5.82430572
 5.82430572 6.922918   6.922918   6.922918   6.922918   6.922918
 6.22977082 6.922918  ]
```

In [48]:
```
custom_result = transform(corpus,task2_sorted_features)
```

```
100%|██████████████████████████████████████████████████| 746/746 [00:00<00:00, 23317.49it/
s]
```

In [51]:
```
custom_result.toarray()
```

Out[51]:
```
array([[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 [ ]: