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.

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 (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]:

```
''' Importing the required packages'''
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from tqdm import tqdm
import os
import numpy as np
from collections import Counter
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy as np


''' created own custom functions for calculation of idf values'''
def cal_idf(corpus_matrix,get_unique_words):
    idf_vector = []
    for ic, word in  enumerate(get_unique_words):
        idf = 1+(np.log((1+len(corpus_matrix))/(1+np.count_nonzero(corpus_matrix[:,i
        idf_vector.append(idf)
    return idf_vector

''' created own custom functions for calculation of tfidf values'''
def cal_tfidf(matrix,get_idf_vector):
    tfidf_mtx = []
    idfVector = get_idf_vector
    for idx, row in enumerate(matrix):
        each_tfidf = []
        for ic,item in enumerate(row):
            tf  = item/len(idfVector)
            idf = idfVector[ic]
            each_tfidf.append(tf*idf)
        tfidf_mtx.append(each_tfidf)
    return tfidf_mtx


''' Fit function is referenced from AAIC refrence notes which returns vocab '''
def fit(dataset):
    ''' For storing the unique words'''
    unique_words = set()
    ''' checking the dataset instance type'''
    if isinstance(dataset, (list,)):
        ''' Looping through each row in dataset'''
        for row in dataset:
            '''getting the words in sentence and ignoring the words which has lenght
            for word in row.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)
        ''' Finally sorting the words and returning it'''
        unique_words = sorted(list(unique_words))
        ''' Finally mapping the words with respect to its respective index'''
        vocab = {j:i for i,j in enumerate(unique_words)}
        return vocab
    else:
        print("you need to pass list of sentance")
```

```
''' Transform function is referenced from AAIC refrence notes which returns corpus m
def transform(dataset,vocab):
    rows = []
    columns = []
    values = []
    ''' checking the dataset instance type'''
    if isinstance(dataset, (list,)):
        ''' Looping through each row in dataset'''
        for idx, row in enumerate(tqdm(dataset)):
            '''Getting word frequency '''
            word_freq = dict(Counter(row.split()))
            ''' For every unique word in the document'''
            for word, freq in word_freq.items():
                if len(word) < 2:
                    continue
                '''we will check if its there in the vocabulary that we build in fit
                   function will return the values, if the key doesn't exits it will
                '''
                col_index = vocab.get(word, -1) # retreving the dimension number of
                # if the word exists
                if col_index !=-1:
                    ''' we are storing the index of the document '''
                    rows.append(idx)
                    ''' we are storing the dimensions of the word '''
                    columns.append(col_index)
                    # we are storing the frequency of the word
                    values.append(freq)
        return csr_matrix((values, (rows,columns)), shape=(len(dataset),len(vocab)))
    else:
        print("you need to pass list of strings")



''' Given corpus '''
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]

''' Getting the vocab'''
vocab = fit(corpus)

''' Getting the unique words'''
unique_words = vocab.keys()
print(unique_words)

'''Getting the corpus matrix'''
corpus_matrix = transform(corpus, vocab).toarray()
# print(corpus_matrix)
```

```
100%|██████████| 4/4 [00:00<00:00, 653.90it/s]


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

In [2]:

```
''' Finally
    1. calculating idf vectop for 9 unique features
    2. calculating tfidf values for 9 features
    3. Normalising the top values for 9 features
    4. Displaying the results
'''
get_idf_vector = cal_idf(corpus_matrix,unique_words)
tfidf_vector     = cal_tfidf(corpus_matrix,get_idf_vector)
tfidf_normalized = normalize(np.array(tfidf_vector), norm='l2', copy=False)
print(get_idf_vector)
print(tfidf_normalized[0])
```

```
[1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0, 1.916
290731874155, 1.916290731874155, 1.0, 1.916290731874155, 1.0]
[0.         0.46979139 0.58028582 0.38408524 0.         0.
 0.38408524 0.         0.38408524]
```

In [3]:

```
''' Displaying the hole tdidf normalized matrix'''
print(tfidf_normalized)
```

```
[[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]]
```

In [4]:

```
## 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 [5]:

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

In [6]:

```
# sklearn feature names, they are sorted in alphabetic order by default.

print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 't
his']
```

In [7]:

```
# Here we will print the sklearn tfidf vectorizer idf values after applying the fit
# After using the fit function on the corpus the vocab has 9 words in it, and each h

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 method.
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.toarray()[0])
```

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

In [10]:

```
# sklearn tfidf values for first line of the above corpus.
# To understand the output better, here we are converting the sparse output matrix t
# Notice that this output is normalized using L2 normalization. sklearn does this by

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

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 [11]:

```
''' Reading the corpus from the file'''
import pickle
with open('cleaned_strings', 'rb') as f:
    cleaned_strings_corpus = pickle.load(f)

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

```
Number of documents in cleaned_strings_corpus =  746
```

In [12]:

```
''' Getting the cleaned vocab from cleaned strings and sorting it '''
cleaned_vocab = fit(cleaned_strings_corpus)
cleaned_unique_words = sorted(list(cleaned_vocab.keys()))
''' Doing the transformation for the unique qords'''
cleaned_strings_corpus_matrix = transform(cleaned_strings_corpus, cleaned_vocab).toa
cleaned_strings_idf_vector = cal_idf(cleaned_strings_corpus_matrix,cleaned_unique_wo
s = pd.Series(cleaned_strings_idf_vector)

''' top values has 1855 times, so idf values for top 50 words will be same '''
s.value_counts()
```

```
100%|██████████| 746/746 [00:00<00:00, 15256.55it/s]
```

Out[12]:

```
6.922918    1855
6.517453     459
6.229771     197
6.006627     104
5.824306      63
5.670155      38
5.536624      26
5.418841      24
5.218170      17
5.313480      17
5.051116      12
5.131159      12
4.671626       9
4.977008       8
4.571543       5
4.480571       4
4.725693       4
4.782852       4
4.908015       4
4.357969       2
4.843476       2
4.032546       2
4.397189       2
4.089705       1
4.119558       1
4.005147       1
3.978479       1
2.718225       1
3.787424       1
4.525023       1
3.952504       1
4.283861       1
2.771878       1
3.627081       1
2.897566       1
4.150329       1
4.182078       1
3.573014       1
4.620333       1
dtype: int64
```

In [13]:

```
''' calculating the top 50 idf values '''

top_fivty_idf_index = np.argsort(np.array(cleaned_strings_idf_vector))[-50:][::-1].t
print(top_fivty_idf_index)
''' Getting the top 50 feature words'''
cleaned_fifty_unique_words = [cleaned_unique_words[i] for i in top_fivty_idf_index]
print(cleaned_fifty_unique_words)


'''  Printing top idf values for words'''
print("\n Printing top 50 idf values & words")
for indx,top_idf in enumerate(top_fivty_idf_index):
    print(cleaned_fifty_unique_words[indx],cleaned_strings_idf_vector[top_idf])

''' Finally
    1. Doing tranform on top 50 features
    2. calculating tfidf values for top 50 features
    3. Normalising the top 50 features
    4. Displaying the first record which has shape of (1,50)
'''
get_fivty_vocab = {}
for indx, item in enumerate(cleaned_fifty_unique_words):
    get_fivty_vocab[item]= indx
top_fivty_corpus_matrix    = transform(cleaned_strings_corpus, get_fivty_vocab).toar
top_fivty_tfidf_vector     = cal_tfidf(top_fivty_corpus_matrix,top_fivty_idf_index)
top_fivty_tfidf_normalized = normalize(np.array(top_fivty_tfidf_vector), norm='l2',
print(top_fivty_tfidf_normalized[0])
top_fivty_idf_index[0]
```

```
100%|██████████| 746/746 [00:00<00:00, 67696.90it/s]

[2885, 1152, 1162, 1158, 1156, 1155, 1154, 1153, 1151, 1165, 1148, 114
6, 1145, 1142, 1141, 1140, 1163, 1166, 1089, 1178, 1185, 1183, 1182, 1
181, 1180, 1179, 1177, 1167, 1176, 1173, 1172, 1171, 1170, 1168, 1138,
1136, 1134, 1102, 1108, 1107, 1106, 1105, 1104, 1103, 1100, 1133, 109
9, 1098, 1097, 1095]
['zombiez', 'havilland', 'hearts', 'heads', 'hbo', 'hayworth', 'haya
o', 'hay', 'hatred', 'heche', 'harris', 'happy', 'happiness', 'hanks',
'hankies', 'hang', 'heartwarming', 'heels', 'gone', 'hero', 'higher',
'hide', 'hes', 'heroism', 'heroine', 'heroes', 'hernandez', 'heist',
'hendrikson', 'helping', 'help', 'helms', 'hellish', 'helen', 'handle
s', 'handle', 'ham', 'grade', 'grates', 'grasp', 'graphics', 'grante
d', 'grainy', 'gradually', 'government', 'halfway', 'gotten', 'gotta',
'goth', 'gosh']

 Printing top 50 idf values & words
zombiez 6.922918004572872
havilland 6.922918004572872
hearts 6.922918004572872
heads 6.922918004572872
hbo 6.922918004572872
hayworth 6.922918004572872
hayao 6.922918004572872
hay 6.922918004572872
hatred 6.922918004572872
heche 6.922918004572872
harris 6.922918004572872
happy 6.922918004572872
happiness 6.922918004572872
```

```
hanks 6.922918004572872
hankies 6.922918004572872
hang 6.922918004572872
heartwarming 6.922918004572872
heels 6.922918004572872
gone 6.922918004572872
hero 6.922918004572872
higher 6.922918004572872
hide 6.922918004572872
hes 6.922918004572872
heroism 6.922918004572872
heroine 6.922918004572872
heroes 6.922918004572872
hernandez 6.922918004572872
heist 6.922918004572872
hendrikson 6.922918004572872
helping 6.922918004572872
help 6.922918004572872
helms 6.922918004572872
hellish 6.922918004572872
helen 6.922918004572872
handles 6.922918004572872
handle 6.922918004572872
ham 6.922918004572872
grade 6.922918004572872
grates 6.922918004572872
grasp 6.922918004572872
graphics 6.922918004572872
granted 6.922918004572872
grainy 6.922918004572872
gradually 6.922918004572872
government 6.922918004572872
halfway 6.922918004572872
gotten 6.922918004572872
gotta 6.922918004572872
goth 6.922918004572872
gosh 6.922918004572872
[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. 0.1
```

Out[13]:

```
2885
```

In [14]:

```
''' Additiion approach for my refrence, you can ignore it, while evluation'''

# cleaned_vocab = fit(cleaned_strings_corpus)
# cleaned_unique_words = sorted(list(cleaned_vocab.keys()))
# cleaned_strings_corpus_matrix = transform(cleaned_strings_corpus, cleaned_vocab).t

# cleaned_strings_idf_vector = []
# for ic, word in  enumerate(cleaned_unique_words):
#     idf = 1+np.log((1+len(cleaned_strings_corpus_matrix))/(1+np.count_nonzero(clea
#     cleaned_strings_idf_vector.append(idf)

# cleaned_strings_idf_vector = np.array(cleaned_strings_idf_vector)
# priority_indexs = np.array(cleaned_strings_idf_vector).argsort()[-50:][::-1].tolis
# # print(priority_indexs)
# cleaned_fifty_unique_words = [cleaned_unique_words[i] for i in priority_indexs]
# print(cleaned_fifty_unique_words)
# print("\n")
# tmp = {}
# for indx, item in enumerate(cleaned_fifty_unique_words):
#     tmp[item]= indx

# # print(tmp)
# cleaned_corpus_matrix    = transform(cleaned_strings_corpus, tmp).toarray()
# cleaned_tfidf_vector     = cal_tfidf(cleaned_corpus_matrix ,cleaned_fifty_unique_w
# cleaned_tfidf_normalized = normalize(np.array(cleaned_tfidf_vector), norm='l2', co
# print(cleaned_tfidf_normalized.shape)



# cleaned_strings_corpus_matrix = transform(cleaned_strings_corpus, vocab).toarray()
# print(cleaned_strings_corpus_matrix)


# tfidf_vector     = cal_tfidf(corpus_matrix)
# tfidf_normalized = normalize(np.array(tfidf_vector), norm='l2', copy=False)
# print(tfidf_normalized[0])
```

Out[14]:

```
' Additiion approach for my refrence, you can ignore it, while evluati
on'
```