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 [80]:

```
## 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 [81]:

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

In [82]:

```
# 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']
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

In [83]:

```
# 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 [84]:

```
# shape of sklearn tfidf vectorizer output after applying transform method.

skl_output.shape
```

Out[84]:

```
(4, 9)
```

In [85]:

```
# 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 [86]:

```
# 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.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 [87]:

```
# 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 scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy
```

In [88]:

```
def fit(data):#Return a list of all unique words
  if isinstance(data,(list,)):
    unique_words=set()
    for i in data:# iteratting through each column
      for j in i.split(' '):# Iterating through each row
        if j in unique_words:# if the word is already present just ignore it
          continue
        unique_words.add(j)#else ADD the words to unique list
    data_words=sorted(list(unique_words))#sort the data in alphabetical Order
    d={j:i for i, j in enumerate(data_words)}#convert all the words into dict as words as key and index as value
    return d #return the dict
  else:
    return 'Please pass a list of sentences'
```

In [89]:

```
vocab=fit(corpus)
print(vocab)
```

```
{'and': 0, 'document': 1, 'first': 2, 'is': 3, 'one': 4, 'second': 5, 'the': 6, 'third': 7, 'this': 8}
```

In [90]:

```
def tf_transform(dataset,vocab):# To get term frequency
    rows = []
    columns = []
    values = []
    if isinstance(dataset, (list,)):
      col=0
      row=0

      total_terms=len(vocab)
      for i in dataset:# This will append the row and column values if they are non zero terms
        l=i.split()
        col=0
        for word in vocab: #iterate through each unique word of a dataset
          if word in l:
            freq=l.count(word)
            rows.append(row)
            columns.append(col)
            values.append(freq/len(l)) # calculate term frequency of a word in a given sentence
          col=col+1
        row=row+1
      return csr_matrix((values,(rows,columns)), shape=(len(dataset),len(vocab)))# return sparse matrix

    else:
        print("you need to pass list of strings")
```

In [91]:

```
TF=tf_transform(corpus,vocab)
print('\n',TF.toarray())
```

```
 [[0.         0.2        0.2        0.2        0.         0.
  0.2        0.         0.2       ]
 [0.         0.33333333 0.         0.16666667 0.         0.16666667
  0.16666667 0.         0.16666667]
 [0.16666667 0.         0.         0.16666667 0.16666667 0.
  0.16666667 0.16666667 0.16666667]
 [0.         0.2        0.2        0.2        0.         0.
  0.2        0.         0.2       ]]
```

In [92]:

```
def idf_transform(corpus,vocab):
  out=[]
  total_docs=len(corpus)
  for i in vocab:# pick each word from  unique_words
    count=0
    for j in corpus: # Go Through each sentencce in the corpus
      if i in j.split():
        count=count+1 # Count number of sentences contain the word j
    temp=math.log((1+total_docs)/(1+count))+1
    out.append(temp)
  return numpy.array(out)
```

In [93]:

```
print(idf_transform(corpus,vocab))
```

```
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
```

In [94]:

```
def tf_idf_transform_fit(dataset):
  vocab=fit(dataset)
  tf=tf_transform(dataset,vocab).toarray()# To get Term frequecy for given data output is sparse matrix
  idf=idf_transform(dataset,vocab)#To get Inverse term frequency for given data  output is sparse matrix
  rows=[]
  columns=[]
  values=[]
  row=0
  for i in tf:
    col=0
    for j,t in zip(i,idf):
      val=j*t  # find TFIDF by multiplying TF and IDf for given term
      if val!=0:
        rows.append(row)
        columns.append(col)
        values.append(val)
      col=col+1
    row=row+1
  return norm(csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab))))# do l2 normalization and return the data in the sparse matrix
  #the sparse matrix must be size of number of documents * number of unique elements
def norm(mat):# apply l2 normalization for the given sparse matrix
  return normalize(mat,norm='l2',axis=1,copy=True,return_norm=False)
```

In [96]:

```
print(tf_idf_transform_fit(corpus).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]]
```

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 [101…

```
# Below is the code to load the cleaned_strings pickle file provided
# Here corpus is of list type
from google.colab import files
uploaded = files.upload()
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))
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving cleaned_strings to cleaned_strings (2)
Number of documents in corpus =  746
```

In [22]:

```
# Write your code here.
# Try not to hardcode any values.
# Make sure its well documented and readble with appropriate comments.
```

In [108…

```
import numpy as np
def top_idf_values_fit(corpus,top=50):
  vocab=fit(corpus)#To get all unique values
  import numpy as np
  idf_values_of_all_unique_words=idf_transform(corpus,vocab)#To get IDF values of all unique words
  idf_sorted_top_50_indicies=np.argsort(list(idf_values_of_all_unique_words))[::-1][:top]#To get the sorted indices of top 50 idf values
  top_50_idf_values=np.take(list(idf_values_of_all_unique_words),idf_sorted_top_50_indicies)#To get top 50 idf values
  top_50_idf_words=np.take(list(vocab),idf_sorted_top_50_indicies)# T get top 50 indices words that has top 50 idf values
  top_50_idf=dict(zip(top_50_idf_words,top_50_idf_values))# creating a dict with words as key and value as values of dict
  return top_50_idf_words,top_50_idf
```

In [109…

```
def tf_idf_transform_fit_top(dataset,top=50):#return a sparse matrix with size of n * 50
  vocab=fit(dataset)
  tf=tf_transform(dataset,vocab).toarray()# To get Term frequecy for given data output is sparse matrix
  vocab,idf=top_idf_values_fit(dataset)#To get Inverse term frequency for given data  output is sparse matrix
  rows=[]
  columns=[]
  values=[]
  row=0
  for i in tf:
    col=0
    for j,t in zip(i,list(idf.values())):
      val=j*t  # find TFIDF by multiplying TF and IDf for given term
      if val!=0:
        rows.append(row)
        columns.append(col)
        values.append(val)
      col=col+1
    row=row+1
  return norm(csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab))))# do l2 normalization and return the data in the sparse matrix
  #the sparse matrix must be size of number of documents * number of unique elements
def norm(mat):# apply l2 normalization for the given sparse matrix
  return normalize(mat,norm='l2',axis=1,copy=True,return_norm=False)
```

In [110…

```
print(tf_idf_transform_fit_top(corpus))
```

```
  (2, 20)	1.0
  (10, 36)	1.0
  (15, 20)	1.0
  (17, 20)	1.0
  (19, 4)	0.4472135954999579
  (19, 23)	0.4472135954999579
  (19, 24)	0.4472135954999579
  (19, 27)	0.4472135954999579
  (19, 32)	0.4472135954999579
  (26, 19)	1.0
  (28, 27)	1.0
  (36, 24)	1.0
  (41, 20)	1.0
  (49, 20)	1.0
  (56, 14)	1.0
  (60, 44)	1.0
  (62, 39)	1.0
  (65, 23)	1.0
  (68, 43)	1.0
  (72, 24)	1.0
  (86, 20)	1.0
  (104, 24)	1.0
  (134, 4)	0.5773502691896257
  (134, 24)	0.5773502691896257
  (134, 47)	0.5773502691896257
  :	:
  (644, 17)	0.2
  (644, 18)	0.2
  (644, 20)	0.8
  (644, 23)	0.2
  (644, 24)	0.2
  (644, 49)	0.2
  (649, 20)	1.0
  (658, 20)	1.0
  (660, 4)	1.0
  (667, 41)	1.0
  (669, 20)	1.0
  (673, 23)	1.0
  (688, 27)	1.0
  (697, 12)	1.0
  (706, 2)	0.7071067811865476
  (706, 20)	0.7071067811865476
  (707, 24)	1.0
  (710, 47)	1.0
  (712, 27)	1.0
  (718, 18)	1.0
  (722, 20)	0.7071067811865475
  (722, 31)	0.7071067811865475
  (725, 19)	1.0
  (726, 26)	1.0
  (738, 25)	1.0
```

In [69]: