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:

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

```
## 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

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

# 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)
```

```
# 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.        ]
```

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

skl_output.shape
```

```
(4, 9)
```

```
# 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
```

```
# 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]]
```

Your custom implementation

```
# 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 as np

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

def IDF(corpus, unique_words):
   idf_dict={}
   N=len(corpus)
   for i in unique_words:
     count=0
     for sen in corpus:
       if i in sen.split():
         count=count+1
       idf_dict[i]=(math.log((1+N)/(count+1)))+1
   return idf_dict

def fit(whole_data):
    unique_words = set()
    if isinstance(whole_data, (list,)):
      for x in whole_data:
        for y in x.split():
          if len(y)<2:
            continue
          unique_words.add(y)
      unique_words = sorted(list(unique_words))
      vocab = {j:i for i,j in enumerate(unique_words)}
      Idf_values_of_all_unique_words=IDF(whole_data,unique_words)
    return vocab, Idf_values_of_all_unique_words
Vocabulary, idf_of_vocabulary=fit(corpus)

print(list(Vocabulary.keys()))

print(list(idf_of_vocabulary.values()))

def transform(dataset,vocabulary,idf_values):
     sparse_matrix= csr_matrix( (len(dataset), len(vocabulary)),
dtype=np.float64)
```

```
     for row  in range(0,len(dataset)):
       number_of_words_in_sentence=Counter(dataset[row].split())
       for word in dataset[row].split():
           if word in  list(vocabulary.keys()):

tf_idf_value=(number_of_words_in_sentence[word]/len(dataset[row].split
()))*(idf_values[word])
               sparse_matrix[row,vocabulary[word]]=tf_idf_value
     #print("NORM FORM\n",normalize(sparse_matrix, norm='l2', axis=1,
copy=True, return_norm=False))
     output =normalize(sparse_matrix, norm='l2', axis=1, copy=True,
return_norm=False)
     return output
final_output=transform(corpus,Vocabulary,idf_of_vocabulary)
print(final_output.shape)

print(final_output.toarray())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third',
'this']
[1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0,
1.916290731874155, 1.916290731874155, 1.0, 1.916290731874155, 1.0]
(4, 9)
[[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]]

/usr/local/lib/python3.7/dist-packages/scipy/sparse/_index.py:84:
SparseEfficiencyWarning: Changing the sparsity structure of a
csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```

Task-2

2. Implement max features functionality:

```
# 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
```

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

def IDF(corpus, unique_words):
   idf_dict={}
   N=len(corpus)
   for i in unique_words:
     count=0
     for sen in corpus:
       if i in sen.split():
         count=count+1
       idf_dict[i]=(math.log((1+N)/(count+1)))+1
   return idf_dict

def fit(corpus):
  vocab=generate_unique_words(corpus)
  idf_values_of_all_unique_words=IDF(corpus, vocab)

top50_idf_value_indices=np.argsort(list(idf_values_of_all_unique_words
.values()))[::-1][:50]


top50_idf_values=np.take(list(idf_values_of_all_unique_words.values())
, top50_idf_value_indices)

top50_idf_words=np.take(list(idf_values_of_all_unique_words.keys()),to
p50_idf_value_indices)

  reduced_idf_of_vocabulary=dict(zip(top50_idf_words,
top50_idf_values))
  reduced_vocabulary={key:value for value,key in
enumerate(list(reduced_idf_of_vocabulary.keys()))}

  return
  reduced_vocabulary, reduced_idf_of_vocabulary

def transform(dataset,vocabulary,idf_values):
     sparse_matrix= csr_matrix( (len(dataset), len(vocabulary)),
dtype=np.float64)
     for row  in range(0,len(dataset)):
       number_of_words_in_sentence=Counter(dataset[row].split())
       for word in dataset[row].split():
           if word in  list(vocabulary.keys()):

tf_idf_value=(number_of_words_in_sentence[word]/len(dataset[row].split
()))*(idf_values[word])
               sparse_matrix[row,vocabulary[word]]=tf_idf_value
     #print("NORM FORM\n",normalize(sparse_matrix, norm='l2', axis=1,
copy=True, return_norm=False))
     output =normalize(sparse_matrix, norm='l2', axis=1, copy=True,
return_norm=False)
     return output
final_output=transform(corpus,Vocabulary,idf_of_vocabulary)
#print(final_output.shape)

print(final_output.toarray())
```

```
[[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.]]

/usr/local/lib/python3.7/dist-packages/scipy/sparse/_index.py:84:
SparseEfficiencyWarning: Changing the sparsity structure of a
csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```