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

nes a word appears in a document, divided by the total
e 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 ...uld need to add following things to your custom ...r:

  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.

Saved successfully!

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.

Saved successfully!

elpful for you to debug the code you write with print
en 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: F
  warnings.warn(msg, category=FutureWarning)
```

```
# After using the fit function on the corpus the vocab has 9 words in it, and each has its id
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
```



```
  (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 m
# 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

```
# Importing required libraries
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

# fitting the tfidf vectorizer on corpus
def fit(dataset):
  unique_words = set();
  if isinstance(dataset,(list,)):
    # looping over each document in the corpus
    for row in dataset:
      # seperating each word from the row
      for word in row.split(" "):
        if len(word) < 2:
          continue
        # adding each word/adjective in list
        unique_words.add(word)
    # sorting unique_words list in ascending order
    unique_words = sorted(list(unique_words))
    # getting dimension of unique_words
    vocabulary = {j:i for i,j in enumerate(unique_words)}
    return vocabulary
  else:
    print("You need to pass list of string")
```



```
# transforming the tfidf vectorizer using tf and idf technique
def transform(dataset,vocabulary):
  rows = []     # rows to add index of document from corpus
  columns = []  # colmuns to add column index
  values = []   # values to append tf-idf values of each words

  if isinstance(dataset,(list,)):
    # looping over each document in the corpus
    for idx,row in enumerate(tqdm(dataset)):
      # creating dictionary for word frequency
      word_freq = dict(Counter(row.split(" ")))
      # getting word along with its freq of each word in row
      for word,freq in word_freq.items():
        if len(word) < 2:
          continue
        # get column index of a word which present in vocabulary
        column_index = vocabulary.get(word,-1)
        # if index not -1, perform tf-idf
        if column_index != -1:
          rows.append(idx)
```

```
            columns.append(column_index)
            # calculate Term Frequency(TF)
            tf = freq/float(len(row))
            # calculate Inverse Document Frequency(IDF)
            idf_ = 1+math.log((1+float(len(dataset))) / float((1+idf(dataset,word))))
            # multiply tf and idf values to get tf-idf score for a word
            values.append((tf) * (idf_))
      # L2 Normalization on output matrix which is sparse matrix
      return normalize(csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocabulary))))
    else:
      print("You need to pass list of string")

  # calculate the IDF value for its words in the given corpus
  def idf(dataset,word):
    term_occur = 0
    for i in dataset:
      # splitting each word from row
      if word in i.split(" "):
        # get occurence of a term
        term_occur += 1
    return term_occur

  # transform the tf-idf vectorizer
  tfidf_vectorizer = transform(corpus,vectorizer)
  print(tfidf_vectorizer)
```

```
     {'and': 0, 'document': 1, 'first': 2, 'is': 3, 'one': 4, 'second': 5, 'the': 6, 'third'
100%|██████████| 141 [00:00<00:00, 6845.05it/s]  (0, 1) 0.46979138557992045
                                                  359
                                                  149
       (0, 6)        0.3840852409148149
       (0, 8)        0.3840852409148149
       (1, 1)        0.6876235979836938
       (1, 3)        0.2810886740337529
       (1, 5)        0.5386476208856763
       (1, 6)        0.2810886740337529
       (1, 8)        0.2810886740337529
       (2, 0)        0.511848512707169
       (2, 3)        0.267103787642168
       (2, 4)        0.511848512707169
       (2, 6)        0.267103787642168
       (2, 7)        0.511848512707169
       (2, 8)        0.267103787642168
       (3, 1)        0.46979138557992045
       (3, 2)        0.5802858236844359
       (3, 3)        0.3840852409148149
       (3, 6)        0.3840852409148149
       (3, 8)        0.3840852409148149
```

Saved successfully!

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

    int idf values for each term in your vocab.
    ır 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.

```
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
```

```
from enum import unique
```

```
def fit_top_50(dataset):
  unique_words_list = []
  idf_values_list = []
  if isinstance(dataset,(list,)):
    # looping over each document in the corpus
    for row in dataset:
      for word in row.split(" "):
        # adding unique word in list
        if len(word) >= 2 and word not in unique_words_list:
          unique_words_list.append(word)
    for u_word in unique_words_list:
      # get idf value for each unique word
      idf_value = 1+(math.log((1+float(len(dataset))) / float(1+idf(dataset,u_word))))
      # store idf value in list
      idf_values_list.append(idf_value)
    # combine unique word with its idf value
    combined_unique_words_with_idf_value = zip(idf_values_list,unique_words_list)
    # sort combined list in descending order with idf value
    sorted_combined_unique_words_with_idf_value = sorted(combined_unique_words_with_idf_value
    sorted_unique_words_list = [element for _, element in sorted_combined_unique_words_with_i
    # get the dimention/vocab of top 50 sorted unique word
    vocab_top_50 = {j:i for i,j in enumerate(sorted_unique_words_list[:50])}
    return vocab_top_50,sorted_combined_unique_words_with_idf_value[:50]
  else:
    print("you need to pass list of strings")
```

Saved successfully!

```
fit_top_50(corpus)
```

```
# chosen 50 word with their idf values
for i,j in enumerate(top_50_word_with_IDF):
  print(j)
```

```
(6.922918004572872, 'zombiez')
(6.922918004572872, 'zillion')
(6.922918004572872, 'yun')
(6.922918004572872, 'youtube')
(6.922918004572872, 'youthful')
(6.922918004572872, 'younger')
(6.922918004572872, 'yelps')
(6.922918004572872, 'yawn')
(6.922918004572872, 'yardley')
(6.922918004572872, 'wrote')
(6.922918004572872, 'writers')
(6.922918004572872, 'wrap')
(6.922918004572872, 'wow')
(6.922918004572872, 'woven')
(6.922918004572872, 'wouldnt')
(6.922918004572872, 'worthy')
(6.922918004572872, 'worthwhile')
```

```
        (6.922918004572872, 'worthless')
        (6.922918004572872, 'worry')
        (6.922918004572872, 'worked')
        (6.922918004572872, 'woo')
        (6.922918004572872, 'wont')
        (6.922918004572872, 'wong')
        (6.922918004572872, 'wondered')
        (6.922918004572872, 'woa')
        (6.922918004572872, 'witticisms')
        (6.922918004572872, 'within')
        (6.922918004572872, 'wise')
        (6.922918004572872, 'win')
        (6.922918004572872, 'wily')
        (6.922918004572872, 'willie')
        (6.922918004572872, 'william')
        (6.922918004572872, 'wild')
        (6.922918004572872, 'wih')
        (6.922918004572872, 'wife')
        (6.922918004572872, 'widmark')
        (6.922918004572872, 'wide')
        (6.922918004572872, 'whoever')
        (6.922918004572872, 'whites')
        (6.922918004572872, 'whine')
        (6.922918004572872, 'whenever')
        (6.922918004572872, 'went')
        (6.922918004572872, 'welsh')
        (6.922918004572872, 'weight')
        (6.922918004572872, 'wedding')
        (6.922918004572872, 'website')
        (6.922918004572872, 'weaving')
        (6.922918004572872, 'wb')
```

Saved successfully!

```
def transform_top_50(dataset,vocab_top_50):
  rows = []
  columns = []
  values = []
  if isinstance(dataset,(list,)):
    # looping over each document in the corpus
    for idx,row in enumerate(tqdm(dataset)):
      word_freq = dict(Counter(row.split(" ")))
      # for each unique word in the document
      for word,freq in word_freq.items():
        if len(word) >= 2:
          # getting dimention of word from vocab that we build iin fit()
          # if word is present in vocab_top_50 it will return index otherwise -1
          dim_word = vocab_top_50.get(word,-1)
          if dim_word != -1:
            rows.append(idx)  # store index of row/document
            columns.append(dim_word)  # store dimention of a word
            # calculate Term Frequency
            tf = freq/float(len(row.split()))
            # calculate IDF value for a word
```

```
            idf_ = 1+(math.log((1+float(len(dataset))) / float(1+idf(dataset,word))))
            # store tf-idf values
            values.append((tf)*(idf_))
          # sparse matrix of if-idf values
          sparseMatrix = csr_matrix((values,(rows,columns)), shape=(len(dataset),len(vocab_to
          # normalize the sparse matrix with l2 normalization
          L2_normalize_output = normalize(sparseMatrix,norm='l2')
    return L2_normalize_output,sparseMatrix
  else:
    print("you need to pass list of strings")

# transform top_50 feature
tf_idf_vectorized_top_50,final_sparseMatrix = transform_top_50(corpus, vocabulary_50)
```

```
    100%|██████████| 746/746 [00:03<00:00, 204.82it/s]
```

```
# sparse matrix in sparse format
final_sparseMatrix[0].toarray()
```

```
    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., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
            0., 0.]])
```

```
# transformed top 50 words with l2 norm
print(tf_idf_vectorized_top_50[0].toarray())
```

Saved successfully!

```
[[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. 0. 0.
  0. 0.]]
```

```
# Clear way to visualize output is to convert to pandas dataframe and then use todense()
tf_idf_output = tf_idf_vectorized_top_50[0]
import pandas as pd
# create dataframe with dense output with index as words and column as tf-idf
tf_idf_df = pd.DataFrame(tf_idf_output.T.todense(), index=vocabulary_50.keys(), columns=['tf-
tf_idf_df.sort_values(by=["tf-idf"],ascending=False)
tf_idf_df.T
```

	zombiez	zillion	yun	youtube	youthful	younger	yelps	yawn	yardley	wrote	...
tf-idf	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

1 rows × 50 columns

✓ 0s completed at 1:00 PM

Saved successfully!