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.

</final>

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.

- TFI: 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:
  $TF(t) = (\text{Number of times term t appears in a document})/(\text{Total number of terms in the document})$.$\$$
- 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:
  $IDF(t) = \log_{e}(\text{Total number of documents})/(\text{number of documents with term t in it})$.$\$$ for numerical stability we will be changing this formula little bit $IDF(t) = \log_{e}(\text{Total number of documents})/(\text{number of documents with term t in it}+1)$.$\$$

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 *ten 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> </final>

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 implementation 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 in lexicographical order
  2. Sklearn formula of idf is different from the standard textbook formula. Here the constant **e** is added to the numerator and denominator of the idf as if an extra document was encountered once more than every term in the collection exactly once, which prevents zero divisions: $IDF(t) = 1+\log_{e}(\text{Total number of documents in collection})/(1+(\text{Number of documents with term t in it}))$.$\$$
  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:
  a. You would have to write both fit and transform methods for your custom implementation of tfidf vectorizer.
  b. Print out the alphabetically sorted voab after you fit your data and check if it has the same or diff of the feature names from sklearn tfidf vectorizer.
  c. Print out the idf values from your implementation and check if its the same as that of sklearns tfidf vectorizer idf values.
  d. Once you get your vocab and idf values to be same as that of sklearns implementation of tfidf vectorizer, proceed to the below steps.
  e. 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
  f. After completing the above steps, print the output of your custom implementation and compare it with sklearns implementation of tfidf vectorizer.
  g. 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. And then dense matrix should contain 1 row and ja columns.

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

```
# Collection of strings 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']

# 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, 9)	0.38408524091481483
  (0, 8)	0.35372139305095475
  (0, 3)	0.46735097643665487
  (0, 6)	0.35372139305095475
  (0, 1)	0.46735097643665487

# sklearn tfidf values for first line of the above corpus
# To understand your output better for above output, check the sparse output matrix in dense matrix and printing
# Notice that this output is normalized using L2 normalization. sklearn does this by default.
print(skl_output[0].toarray())

[[0.         0.46735097 0.52832083 0.38408524 0.         0.
  0.         0.35372139 0.        ]]
```

Your custom implementation

```
# Write your code here.
# Make sure its well documented and readable with appropriate comments.
# Compare your output with that of sklearn tfidf vectorizer.
# You are not supposed to use any other library apart from the ones given below

from collections import Counter
from math import log
from time import sleep
from scipy.sparse import csr_matrix
import math
from sklearn.preprocessing import normalize
import numpy
import pandas
import operator
wordtags = filter(lambda x:len(x)>0, [w.strip("\'\"?,.:;-!") for w in
Corpus.lower().split(' ')])
print wordtags
import math

def fit(Corpus):
    # print("In the fit well") # set the word_dict as empty set
    if isinstance(Corpus,(list,)):# check for the corpus type
        for row in Corpus:
            # print("In the for loop") # Splits the each word in the row
            if len(row) == 0 | len(row.strip()) == 0:# ignore the empty lines
                continue
            # word_list = re.sub('[^A-Za-z0-9]+', ' ', row).split(' ')
            wordtags = filter(lambda x:len(x)>0, [w.strip("\'\"?,.:;-!") for w in
row.lower().split(' ')])
            # print("Word list",word_dict) # checks the dict of word with their index value
            return True,wordtags
    else:
        print("Please provide the data in list")

def transform(corpus, vocab):
    # print("In the fit")
    columns = 0
    values = []
    if isinstance(corpus, (list,)):
        for idx, row in enumerate(enumerate(corpus,1),1):
            # print("Inside the for loop for the word and freq as a value, for every unique word in the d
            word_freq = Counter(row[1])
            for word, freq in word_freq.items():# For each unique word in the review.
                columns = idx
                values.append(
                    (idx,vocab[word],freq,
                    # we will check if the term is the vocabulary that we build in fit function
                    # and if it is then we will store its values, if key doesn't exists it return -1
                    vocab.get(word, -1)
                    if vocab.get(word, -1) != -1
                    # if the word is present, fetching, index of the document
                    # and the word index, frequency, dimension of the word
                    # values.append(freq) #printing, for debugging
                    ))# end of for loop
        return csr_matrix(values, (rows,columns), shape=(len(corpus),len(vocab)))
    else:
        print("Please provide the data in list")

Corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
vocab = fit(Corpus)
print(vocab)
print(transform(Corpus, vocab).toarray())
```

```
[redacted] 4 k (20:00:00)
400B 200B/s
['and', 'is', 'document', 'is', 'first', 'is', 'is', 'one', 'is', 'second', 'is', 'the', 'is', 'third', 'is', 'this', 'is']
[ 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0
 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0]
```

```
# calculating the term_frequency

def term_frequency():
    t_frequency=[]
    val = transform(Corpus, vocab).toarray() # calling the function to get the corpus
    for row in val:
        for row in range(len(val)):# for the sub row in the corpus
            temp = []
            for col in range(len(val[row])):# for each word in the row
                value = val[row][col]#accessing the value
                temp.append(value)
            t_frequency.append(temp)
    return(t_frequency)

# calculating the inverse_document_frequency:

def inverse_document_frequency():
    doc_freq = []
    idf_dict = {}
    vocab = {}
    val = transform(Corpus, vocab).toarray()
    vocab_v = vocab
    for word in range(len(vocab_v)):
        count = 0
        for doc in range(len(val)):# finding the word in all rows of the corpus
            if val[doc][word]!=0:
                count += 1
        doc_freq.append(count)# storing, the frequency of word in the corpus for all row
    for freq in range(len(doc_freq)):
        # print("Inside the for loop", freq)
        dc = float(len(val))/(doc_freq[freq])
        value = lambda x:log(x)
        idf_freq.append(value)
        return(idf_freq)

print(inverse_document_frequency())
```

```
[redacted] 4 k (2
400B 200B/s
[1.9162907318741551, 1.2231435513142097, 1.5108256237659907, 1.0, 1.9162907318741551, 1.9162907318741551, 1.0, 1.9
162907318741551, 1.0]
```

```
# calculating the tf_idf

def tf_idf():
    tfidf = 0.0
    tf = term_frequency() # calling the function to get the tf value
    idf = inverse_document_frequency() # calling the function to get the idf value
    for i,j in zip(tf,idf):
        # print("Inside for loop")
        tfidf = 0.0
        for p in range(len(tf)):
            tfidf = tf[p]*idf[p]
            tf_idf.append(tfidf)
            tfidf = 0.0*tf[p]
    return normalize(tf_matrix(tfidf,(r,c)),norm='l2')
```

```
print(tf_idf())[0]
```

```
[redacted] 4 k (20:00:00),
400B 200B/s
[redacted] 4 k (20:00:00),
400B 200B/s
400B 0B/s[redacted]
(0, 8)	0.46735097643665
(0, 3)	0.35372139305095
(0, 6)	0.46735097643665
(0, 1)	0.35372139305095
(0, 0)	0.38408524091481
  (0, 9)	0.46735097643665
  (0, 8)	0.35372139305095
  (0, 3)	0.46735097643665
  (0, 6)	0.35372139305095
  (0, 1)	0.38408524091481
```

```
print(tf_idf())[0].toarray()
```

```
[redacted] 4 k (20:00:00),
400B 200B/s
[redacted] 4 k (20:00:00),
400B 200B/s
400B 0B/s[redacted]
[[0.         0.46735097 0.52832083 0.38408524 0.         0.
  0.         0.35372139 0.        ]
 [redacted]
```

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 (or terms with top tf-idf scores.
- This task is similar to your previous task, just that here your vocabulary is limited to only top (n features number) based on their tfidf values. Basically your output will have exactly *n* columns and the number of rows will depend on the number of documents you have in your corpus.
- Here you will be given a pickle file, with file name **sklearn_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:
  a. 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 go as described above.
  b. Now sort your vocab based in descending order of tfidf values and print out the words in the sorted vocab after you fit your data.
  Here you should be able to print only top n terms in your vocab. And make sure to print tfidf values for each term in your vocab.
  c. 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
  d. 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 *n* row and ja columns.

```
# 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.
# Top max_features vocabulary.
# Make sure its well documented and readable with appropriate comments.

vocab = fit(Corpus) # calling the function to get the vocab
Vocab = list(fit(Corpus))

idf = inverse_document_frequency() # calling the function to get the idf value
```

```
[redacted] 746/746 [00:00:00],
8012/5000[%]
```

```
idf_Vocab = dict(zip(Vocab,idf)) # creating dict idf_vocab_word as key, value respectively

sorted_idf_Vocab = dict(sorted(idf_Vocab.items(), key=operator.itemgetter(1),reverse=True)) # sort the dict an
```

```
top_tfidf = dict(idf_Vocab.items())[:501] # getting the top-50 idf value with respect to the words
print(top_tfidf)

{'realizing': 6.922918047047921, 'abandoned': 6.922918047047921, 'abroad': 6.922918047047921, 'abstract': 6.9229
18047047921, 'accessible': 6.922918047047921, 'accessibility': 6.922918047047921, 'accessible': 6.922918047047921,
'accomplishment': 6.922918047047921, 'accomplishments': 6.922918047047921, 'accounts': 6.922918047047921, 'accurately': 6.9229
18047047921, 'achieve': 6.922918047047921, 'achieved': 6.922918047047921, 'achieving': 6.922918047047921, 'acknowledge': 6.922918
047047921, 'adapt': 6.922918047047921, 'adder': 6.922918047047921, 'admire': 6.922918047047921, 'admiring': 6.922918047047921,
'adv': 6.922918047047921, 'advised': 6.922918047047921, 'adviser': 6.922918047047921, 'advisers': 6.922918047047921, 'advisor':
6.922918047047921, 'advisors': 6.922918047047921, 'affected': 6.922918047047921, 'affecting': 6.922918047047921, 'afterwards':
6.922918047047921, 'agreed': 6.922918047047921, 'agreement': 6.922918047047921, 'agreements': 6.922918047047921, 'agreeing':
6.922918047047921, 'ahead': 6.922918047047921, 'aimed': 6.922918047047921, 'akamai': 6.922918047047921, 'akamai': 6.922918047047921,
'alarmed': 6.922918047047921, 'alarming': 6.922918047047921, 'alarming': 6.922918047047921, 'alarming': 6.922918047047921,
'alone': 6.922918047047921, 'amazed': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing':
6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.9
22918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918
047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921,
'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing': 6.922918047047921, 'amazing':
6.922
```