

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:

$$TF(t) = \frac{\text{Number of times term } t \text{ appears in a document}}{\text{Total number of terms in the document}}.$$

- **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:

$$IDF(t) = \log_e \frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}}.$$

for numerical stability we will be changing this formula
 little bit $IDF(t) = \log_e \frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it} + 1}.$

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 sklearn's 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 generated from idf sorted 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.

$$IDF(t) = 1 + \log_e \frac{1 + \text{Total number of documents in collection}}{1 + \text{Number of documents with term } t \text{ in it}}$$
 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 sklearn's tfidf vectorizer idf values.
 4. Once you get your voacb and idf values to be same as that of sklearn's 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 sklearn's 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 sklearn's 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 sklearn's 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 [9]:

```
## 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 [10]:

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

In [11]:

```
# 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']
```

In [13]:

```
# 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 [14]:

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

```
skl_output.shape
```

Out[14]:

```
(4, 9)
```

In [7]:

```
# sklearn tfidf values for first line of the above corpus.  
# Here the output is a sparse matrix  
  
print(sk1_output[0])
```

```
(0, 8)      0.38408524091481483  
(0, 6)      0.38408524091481483  
(0, 3)      0.38408524091481483  
(0, 2)      0.5802858236844359  
(0, 1)      0.46979138557992045
```

In [15]:

```
# 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(sk1_output[0].toarray())
```

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

Your custom implementation

In [17]:

```
from tqdm import tqdm
from collections import Counter
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy

# fit function accepts only list of sentences
def fit(dataset):
    #initialize empty set
    unique_words = set()
    non_unique_words = []
    # check if dataset is list type or not
    if isinstance(dataset, (list,)):
        #for each row in dataset
        for row in dataset:
            #for each word in dataset
            for word in row.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)
                non_unique_words.append(word)
            unique_words = sorted(list(unique_words))
            non_unique_words = sorted(list(non_unique_words))
        #Vocab dict of unique words
        vocab = {j:i for i,j in enumerate(unique_words)}
        vocab1=vocab
        #initialize vocab1 dict,vocab1 dict will hold number of documents a word occurs
```

```

for k in vocab1:
    vocab1[k]=0.0
#IDF dict
for k in vocab1:
    vocab1[k]=0.0
#populate vocab1 dict
for row in dataset:
    unique_words1=set()
    for word in row.split(" "):
        unique_words1.add(word)
    for i in unique_words1:
        vocab1[i]=vocab1[i]+1.0
#calculate IDF value and populate IDF dict
idf_dict=vocab1
len_u=int(len(dataset))
for i in unique_words:
    idf_dict[i] = 1 + numpy.log((len_u+1)/(vocab1[i]+1))
#vocab dict will have all unique words with their index
vocab = {j:i for i,j in enumerate(unique_words)}
#Transform function
#calls transform function to calculate TF-IDF
output=transform(dataset,vocab,idf_dict).toarray()
print("Output row 1: \n {}".format(output[0]))
print("Output all rows: \n {}".format(output))
return
else:
    print("you need to pass list of sentence")

#Transform function calculates the sparse TFIDF vector of list dataset
def transform(dataset,vocab,idf_dict):
    rows = []
    columns = []
    values = []
    if isinstance(dataset, (list,)):
        for idx, row in enumerate(tqdm(dataset)):
            # the line below will return a dict type object where key is the word and values is its frequency, {word:frequency}
            word_freq = dict(Counter(row.split()))
            for word, freq in word_freq.items(): # for each unique word.
                if len(word) < 2:
                    continue
                # we will check if it is there in the vocab that we build in fit() function
                # dict.get() function will return the values, if the key doesn't exist it will return -1
                col_index = vocab.get(word, -1)
                # 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 TFIDF value of the word
                    values.append(freq/len(word_freq)*idf_dict[word])
            #Csr matrix calculates the sparse matrix, normalize sklearn method
            return normalize(csr_matrix((values, (rows,columns)), shape=(len(dataset),len(vocab))),norm='l2')

corpus = [
    'this is the first document',
    'this document is the second document',

```


In [1]:

```
# 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

In [6]:

```

import pickle
from tqdm import tqdm
from collections import Counter
from scipy.sparse import csr_matrix
import math
import operator
from sklearn.preprocessing import normalize
import numpy

# fit function accepts only list of sentences
def fit(dataset):
    #initialize empty set
    unique_words = set()
    non_unique_words = []
    # check if dataset is list type or not
    if isinstance(dataset, (list,)):
        #for each row in dataset
        for row in dataset:
            #for each word in dataset
            for word in row.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)
                non_unique_words.append(word)
            unique_words = sorted(list(unique_words))
            non_unique_words = sorted(list(non_unique_words))
        #Vocab dict of unique words
        vocab_task2 = {j:i for i,j in enumerate(unique_words)}

        vocab1_task2=vocab_task2
        #initialize vocab1 dict,vocab1 dict will hold number of documents a word occurs
        for k in vocab1_task2:
            vocab1_task2[k]=0.0
        #IDF dict
        for k in vocab1_task2:
            vocab1_task2[k]=0.0
        #populate vocab1 dict
        for row in dataset:
            unique_words1=set()
            for word in row.split(" "):
                unique_words1.add(word)
            for i in unique_words1:
                d_unique =vocab1_task2.get(i, -1)
                if d_unique!=-1:

```

```

vocab1_task2[i]=vocab1_task2[i]+1

#calculate IDF value and populate IDF dict
idf_dict=vocab1_task2
len_u=int(len(dataset))
for i in unique_words:
    idf_dict[i] = 1 + numpy.log((len_u+1)/(vocab1_task2[i]+1))
#sort IDF values in descending order and pick top 50 values
sorted_idf = sorted(idf_dict.items(), key=operator.itemgetter(1), reverse=True)

[:50]

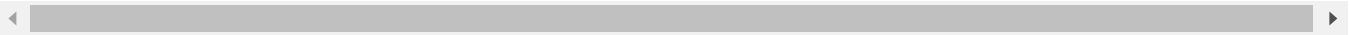
#build IDF dictionary
idf_key_list=[]
idf_value_list=[]
for i in range(len(sorted_idf)):
    idf_key_list.append(sorted_idf[i][0])
    idf_value_list.append(sorted_idf[i][1])
sorted_idf_dict = dict(zip(idf_key_list,idf_value_list))

vocab_task2=sorted_idf_dict
#vocab dict will have all unique words with their index
vocab_task2 = {j:i for i,j in enumerate(idf_key_list)}
print("Vocab: \n\n {}".format(vocab_task2))
print("IDF of top 50 features: \n\n {}".format(sorted_idf_dict))
#Transform function
#calls tranform function to calculate TF-IDF
print(transform(dataset,vocab_task2,sorted_idf_dict))
output=transform(dataset,vocab_task2,sorted_idf_dict).toarray()
print("Output all rows: \n {}".format(output))
print(numpy.shape(output))
return
else:
    print("you need to pass list of sentence")

#Transform function calculates the sparse TFIDF vector of List dataset
def transform(dataset,vocab_task2,idf_dict_task2):
    rows = []
    columns = []
    values = []
    if isinstance(dataset, (list,)):
        for idx, row in enumerate(tqdm(dataset)):
            # the line below will return a dict type object where key is the word and v
            alues is its frequency, {word:frequency}
            word_freq = dict(Counter(row.split()))
            for word, freq in word_freq.items(): # for each unique word.
                if len(word) < 2:
                    continue
                # we will check if it is there in the vocab that we build in fit() func
                tion
                # dict.get() function will return the values, if the key doesn't exists
                it will return -1
                col_index = vocab_task2.get(word, -1)
                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 TFIDF value of the word
                    values.append(freq/len(word_freq)*idf_dict_task2[word])
            #Csr matrix calculates the sparse matrix, normalize sklearn method
            return normalize(csr_matrix((values, (rows,columns)), shape=(len(dataset),len(vocab
_task2)))),norm='l2')

```

```
with open('cleaned_strings', 'rb') as f:  
    corpus = pickle.load(f)  
fit(corpus)
```



```
{ 'aailiyah': 0, 'abandoned': 1, 'abroad': 2, 'abstruse': 3, 'academy': 4,
'accents': 5, 'accessible': 6, 'acclaimed': 7, 'accolades': 8, 'accurate':
9, 'accurately': 10, 'achille': 11, 'ackerman': 12, 'actions': 13, 'adams':
14, 'add': 15, 'added': 16, 'admins': 17, 'admiration': 18, 'admitted': 19,
'adrift': 20, 'adventure': 21, 'aesthetically': 22, 'affected': 23, 'afflec
k': 24, 'afternoon': 25, 'aged': 26, 'ages': 27, 'agree': 28, 'agreed': 29,
'aimless': 30, 'aired': 31, 'akasha': 32, 'akin': 33, 'alert': 34, 'alike':
35, 'allison': 36, 'allow': 37, 'allowing': 38, 'alongside': 39, 'amateuris
h': 40, 'amaze': 41, 'amazed': 42, 'amazingly': 43, 'amusing': 44, 'amust':
45, 'anatomist': 46, 'angel': 47, 'angela': 48, 'angelina': 49}
```

```
{'aailiyah': 6.922918004572872, 'abandoned': 6.922918004572872, 'abroad': 6.922918004572872, 'abstruse': 6.922918004572872, 'academy': 6.922918004572872, 'accents': 6.922918004572872, 'accessible': 6.922918004572872, 'acclaimed': 6.922918004572872, 'accolades': 6.922918004572872, 'accurate': 6.922918004572872, 'accurately': 6.922918004572872, 'achille': 6.922918004572872, 'ackerman': 6.922918004572872, 'actions': 6.922918004572872, 'adams': 6.922918004572872, 'add': 6.922918004572872, 'added': 6.922918004572872, 'admins': 6.922918004572872, 'admiration': 6.922918004572872, 'admitted': 6.922918004572872, 'adrift': 6.922918004572872, 'adventure': 6.922918004572872, 'aesthetically': 6.922918004572872, 'affected': 6.922918004572872, 'affleck': 6.922918004572872, 'afternoon': 6.922918004572872, 'aged': 6.922918004572872, 'ages': 6.922918004572872, 'agree': 6.922918004572872, 'agreed': 6.922918004572872, 'aimless': 6.922918004572872, 'aired': 6.922918004572872, 'akas ha': 6.922918004572872, 'akin': 6.922918004572872, 'alert': 6.922918004572872, 'alike': 6.922918004572872, 'allison': 6.922918004572872, 'allow': 6.922918004572872, 'allowing': 6.922918004572872, 'alongside': 6.922918004572872, 'amateurish': 6.922918004572872, 'amaze': 6.922918004572872, 'amazed': 6.922918004572872, 'amazingly': 6.922918004572872, 'amusing': 6.922918004572872, 'amust': 6.922918004572872, 'anatomist': 6.922918004572872, 'angel': 6.922918004572872, 'angela': 6.922918004572872, 'angelina': 6.922918004572872}
```

```
100%|██████████| 746/746 [00:00<00:00, 39233.51i  
t/s]
```


Output all rows:

```
[[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.]]  
(746, 50)
```

In [0]: