Assignment_3_Completed

July 22, 2022

1 Assignment

What does tf-idf do ? TF-IDF converts a document to a vector each vector has a lenght which is equal to the lenght of the vocublary and each element of that vector is the product of the TF(Term Frequency) * IDF (Inverse Document Frequency).

Important thing to keep in mind is that TF is document dependent but IDF is calculated over all the documents (corpus).

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.

1.1 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:

Sklearn has its vocabulary generated from idf sroted in alphabetical order

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.

$IDF(t) = 1 + \log_e \frac{1 + \text{Total number of documents in collection}}{1+\text{Number of documents with term t in it}}$.

Sklearn applies L2-normalization on its output matrix.

The final output of sklearn tfidf vectorizer is a sparse matrix.

```
<br>
<li>Steps to approach this task:
<ol>
    <li> You would have to write both fit and transform methods for your custom implementation
    <li> Print out the alphabetically sorted voacb after you fit your data and check if its the
    <li> Print out the idf values from your implementation and check if its the same as that of
    <li> Once you get your voacb and idf values to be same as that of sklearns implementation c
    <li> Make sure the output of your implementation is a sparse matrix. Before generating the
    <li> After completing the above steps, print the output of your custom implementation and c
    <li> To check the output of a single document in your collection of documents,  you can con
    </ol>
</li>
<br>
```

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.

1.1.1 Corpus

[1]:
```
## 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',
]
```

1.1.2 SkLearn Implementation

[2]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

[3]:
```
# sklearn feature names, they are sorted in alphabetic order by default.

print(vectorizer.get_feature_names_out())
```

```
['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']
```

[4]:
```
# 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.        ]
```

[5]:
```
# shape of sklearn tfidf vectorizer output after applying transform method.

skl_output.shape
```

[5]:
```
(4, 9)
```

[6]:
```
# sklearn tfidf values for first line of the above corpus.
# Here the output is a sparse matrix
print(type(skl_output))
print(skl_output[0])
```

```
<class 'scipy.sparse.csr.csr_matrix'>
  (0, 8)        0.38408524091481483
  (0, 6)        0.38408524091481483
  (0, 3)        0.38408524091481483
  (0, 2)        0.5802858236844359
  (0, 1)        0.46979138557992045
```

[7]:
```
print(skl_output[1])
```

```
  (0, 8)        0.281088674033753
  (0, 6)        0.281088674033753
  (0, 5)        0.5386476208856763
  (0, 3)        0.281088674033753
  (0, 1)        0.6876235979836938
```

[8]:
```
# 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]]
```

1.1.3 Your custom implementation

[9]:
```
# 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
# alphabatical ordeing of imports
from collections import Counter
from tqdm import tqdm
from scipy.sparse import csr_matrix
```

```
from sklearn.preprocessing import normalize

import math
import operator
import numpy
# 2 line space
# private variable: a__
```

[10]:
```
class TfIdf:
    """

    methods breif
    """

    def __init__(self,corpus,top=None,show=False):
        self.show = show # Flag to print details of all the step not to be use␣
 ↪on big corpus
        self.top = top # For selecting top k idf values If none then use all␣
 ↪values
        self.corpus = corpus # List of docs
        self.uniqueWords = None # Alphabetically sorted list of unique words
        self.idfMap = None # Word to idf value hastable
        self.tfidfVector = None # Sparce Matrix of Vector representation of␣
 ↪Docs


    def sortDictAndPickTopK(self,idfMap):
        """
        idfMap: dict

        sorts the dict and pick top K values for it

        """
        idf = list()
        for key,val in idfMap.items():
            idf.append([key,val])
        sortedIdf = sorted(idf,key = lambda x:x[1],reverse = True)[:self.top]
        sorted_idfMap = dict()
        new_uniqueWords = list()# Limit unique word to self.top
        for key,val in sortedIdf:
            sorted_idfMap[key]=val
            new_uniqueWords.append(key)
        self.idfMap = sorted_idfMap
        self.uniqueWords= new_uniqueWords


    def get_uniqueWords(self):
```

```
        """
        Extract Uniuge Words from the corpus
        """
        uniqueWords = set()
        for doc in tqdm(self.corpus,desc = "Unique Word "):
            for word in doc.split():
                uniqueWords.add(word)

        self.uniqueWords = sorted(list(uniqueWords))# Alphabetically sorted
        if self.show:
            print("Unique Words : \n",self.uniqueWords,"\n\n")
        return self.uniqueWords

    def get_idf(self):
        """
        Calculate idf values
        """
        total_doc = len(self.corpus)# No of docs
        idfMap = dict()
        self.get_uniqueWords()# fetched all unique words
        for uword in tqdm(self.uniqueWords,desc = 'Idf '):
            count=0# No of docs which contains the unique word uword
            for doc in self.corpus:
                if uword in doc.split():
                    count+=1
            idfMap[uword] = 1 + math.log((1 + total_doc)/(1+count)) # idf value␣
↪for word (uword)
        if self.top is not None:
            # sort for top idx values
            self.sortDictAndPickTopK(idfMap)

        else:
            self.idfMap = idfMap

        if self.show:
            print("Idf Values in where Words are alphabetically Sorted: \n␣
↪",list(idfMap.values()),"\n\n")

    def normalize(self,vec,norm):
        """
        vec : Input Vector
        norm: Which norm to use
        """
        div = sum([ val**norm for val in vec])**(1.0/norm)
        return [val/div for val in vec]

    def fit(self):
```

```
        #learn idf
        self.get_idf()

    def transform(self):
        #tranform each doc to vector
        row = list()
        col = list()
        values = list()

        for idxr,doc in tqdm(enumerate(self.corpus),desc="Vectorizer"):
            doc_words = doc.split()
            word_freq = dict(Counter(doc_words))
            if self.show:
                print("Frequency : \n",word_freq)
            tempval = list()# list of vectors which are not normalized
            for idxc,uword in enumerate(self.uniqueWords):# No efficent
                if (uword in doc_words):
                    row.append(idxr)
                    col.append(idxc)
                    tf = word_freq[uword]/len(doc_words) # tf
                    tempval.append(self.idfMap[uword]*tf)# tf idf

                    if self.show:
                        print("Uword : ",uword)
                        print("row : ",idxr)
                        print("col : ",idxc)
                        print("tf : ",tf)
                        print("tfidf : ",self.idfMap[uword]*tf)

            values.extend(self.normalize(tempval,2))# using norm 2 to normalize
        #Creating a sparce Matrix
        self.tfidfVector = csr_matrix((values, (row, col)), shape=(len(self.
↪corpus), len(self.uniqueWords)) )
```

This is equivalent to the above array which we got from skLearn [1.91629073 ,1.22314355 ,1.51082562 ,1. ,1.91629073 ,1.91629073 ,1. ,1.91629073 ,1.]

[11]:
```
tfidf =  TfIdf(corpus,top=None)
tfidf.fit()
tfidf.transform()
```

```
Unique Word : 100%|
              | 4/4 [00:00<00:00, 3443.60it/s]
Idf : 100%|
              | 9/9 [00:00<00:00, 9378.57it/s]
Vectorizer: 4it [00:00, 12865.96it/s]
```

[12]:
```
print(tfidf.tfidfVector.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]]
```

1.2 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:

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.

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.

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

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.

```
</li>
<br>
```

[13]:
```
# 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:
    new_corpus = pickle.load(f)
```

```
# printing the length of the corpus loaded
print("Number of documents in corpus = ",len(new_corpus))
```

```
Number of documents in corpus =  746
```

[14]:
```
# Write your code here.
# Try not to hardcode any values.
# Make sure its well documented and readble with appropriate comments.
```

[15]:
```
tfidf =  TfIdf(new_corpus)
tfidf.fit()
```

```
Unique Word : 100%|
            | 746/746 [00:00<00:00, 194622.80it/s]
Idf : 100%|
            | 2897/2897 [00:01<00:00, 1830.01it/s]
```

[16]:
```
print(f'The number of unique words in the coupus {len(tfidf.uniqueWords)}')
```

```
The number of unique words in the coupus 2897
```

[17]:
```
import numpy as np
tfidfVal = np.array(list(tfidf.idfMap.values()))
```

[18]:
```
print("MIN    : ",np.min(tfidfVal),"\nMAX    : ",np.max(tfidfVal))
print("MEDIAN : ",np.median(tfidfVal))
```

```
MIN    :  2.7182253851819063
MAX    :  6.922918004572872
MEDIAN :  6.922918004572872
```

As the Max and Median values are same it is safe to say that atleast 50% of the words are rare

[19]:
```
tfidf.transform()
```

```
Vectorizer: 746it [00:00, 1844.57it/s]
```

Dimensions of tf-idf vectors using all vocab

[20]:
```
print(tfidf.tfidfVector.shape)
```

```
(746, 2897)
```

The Above are vectors when using all the words of the vocab

TF-IDF using top only 50 rare words

[21]:
```
tfidf =  TfIdf(new_corpus,top=50)# Now we will use only top 50
tfidf.fit()
```

```
Unique Word : 100%|
            | 746/746 [00:00<00:00, 127780.08it/s]
Idf : 100%|
            | 2897/2897 [00:01<00:00, 1877.25it/s]
```

Top 50 IDX values

[22]:
```
for idx,word in enumerate(tfidf.idfMap.keys()):
    if(idx<50):
        print(f"{idx+1} --> {word} IDF Value {tfidf.idfMap[word]}")
```

```
1 --> aailiyah IDF Value 6.922918004572872
2 --> abandoned IDF Value 6.922918004572872
3 --> abroad IDF Value 6.922918004572872
4 --> abstruse IDF Value 6.922918004572872
5 --> academy IDF Value 6.922918004572872
6 --> accents IDF Value 6.922918004572872
7 --> accessible IDF Value 6.922918004572872
8 --> acclaimed IDF Value 6.922918004572872
9 --> accolades IDF Value 6.922918004572872
10 --> accurate IDF Value 6.922918004572872
11 --> accurately IDF Value 6.922918004572872
12 --> achille IDF Value 6.922918004572872
13 --> ackerman IDF Value 6.922918004572872
14 --> actions IDF Value 6.922918004572872
15 --> adams IDF Value 6.922918004572872
16 --> add IDF Value 6.922918004572872
17 --> added IDF Value 6.922918004572872
18 --> admins IDF Value 6.922918004572872
19 --> admiration IDF Value 6.922918004572872
20 --> admitted IDF Value 6.922918004572872
21 --> adrift IDF Value 6.922918004572872
22 --> adventure IDF Value 6.922918004572872
23 --> aesthetically IDF Value 6.922918004572872
24 --> affected IDF Value 6.922918004572872
25 --> affleck IDF Value 6.922918004572872
26 --> afternoon IDF Value 6.922918004572872
27 --> aged IDF Value 6.922918004572872
28 --> ages IDF Value 6.922918004572872
29 --> agree IDF Value 6.922918004572872
30 --> agreed IDF Value 6.922918004572872
31 --> aimless IDF Value 6.922918004572872
32 --> aired IDF Value 6.922918004572872
33 --> akasha IDF Value 6.922918004572872
34 --> akin IDF Value 6.922918004572872
35 --> alert IDF Value 6.922918004572872
36 --> alike IDF Value 6.922918004572872
37 --> allison IDF Value 6.922918004572872
```

```
38 --> allow IDF Value 6.922918004572872
39 --> allowing IDF Value 6.922918004572872
40 --> alongside IDF Value 6.922918004572872
41 --> amateurish IDF Value 6.922918004572872
42 --> amaze IDF Value 6.922918004572872
43 --> amazed IDF Value 6.922918004572872
44 --> amazingly IDF Value 6.922918004572872
45 --> amusing IDF Value 6.922918004572872
46 --> amust IDF Value 6.922918004572872
47 --> anatomist IDF Value 6.922918004572872
48 --> angel IDF Value 6.922918004572872
49 --> angela IDF Value 6.922918004572872
50 --> angelina IDF Value 6.922918004572872
```

[23]:
```
tfidf.transform()
```

```
Vectorizer: 746it [00:00, 46783.15it/s]
```

Dimensions of tf-idf vectors using top-50 rare words

[24]:
```
print(tfidf.tfidfVector.shape)
```

```
(746, 50)
```

Why Normalize ?

https://www.quora.com/What-is-the-benefit-of-normalization-in-the-tf-idf-algorithm

The impact of having the word 'cat' 10 times in document1(with 1000 words) is same as having having the same word 'cat ' 5 times in document2 (with 500 words). If I dont normalise the documents are very distant because lets say we have exactly matching 500 words but the remaning 500 words will make the document vector stracy to much so when I vectorise in the metric space but since I need them to be near , I will normalise.

This would make all the vectors independent of the lenght.

Benefits of Normalizing in tf-idf algorithm depends on the context.

If you are using cosine similarity to find the document similarity and calculating the weightage of vectors using the tf-idf then if does not matter if you normalize or not i.e normalization will have no benefit

If you are using euclidean or any other similarity measure then normalization helps in incorporating the document length in your similarity measure which otherwise would not be incorporated.

[ ]:
```

```