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. </p>

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:
  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.
  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.
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 [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',
]
```

SkLearn Implementation

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

In [3]:
```
vectorizer.fit(corpus)
```

Out[3]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=None,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

In [4]:
```
# 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 [5]:
```
# 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 val
ue.

print (vectorizer.idf_)
```

```
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
```

In [6]:
```
# shape of sklearn tfidf vectorizer output after applying transform method.
skl_output.shape
```

Out[6]:
```
(4, 9)
```

In [7]:
```
# 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
```

In [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[0].toarray())
```

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

Your custom implementation

In [2]:
```
# 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

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

## I have tried to make a class : myTfidf and various methods inside it to achieve the functionali
ty
## similiar TfidfVectorizer in sklearn

class myTfidf:
    def __init__(self):
        pass


    def my_fit(self,corpus):
        self.corpus = corpus


    def my_get_feature_names(self):
        '''This method returns all unique words in the corpus with
        length greater than 2. Words are sorted alphabetically'''
        self.unique_words = set() #empty set
        if isinstance(self.corpus,(list,)):
            for row in (self.corpus):
                for word in (row.split(' ')):
                    if (len(word)) < 2 :
                        continue
                    self.unique_words.add(word)
            sorted_words = sorted(list(self.unique_words))
            return sorted_words
        else:
            print("you need to pass list of sentance")


    def my_idf_(self):
        '''This method calculates IDF of words using the formula mentioned in the sklearn document
ation'''
        sorted_words = self.my_get_feature_names()  ##Calling my_get_feature_names() method to get
sorted words
        self.idf_list = []    #empty list
        self.count_list = []  #empty list
        self.idf_dict = {}    #empty dictionary
        self.word_doc = {}    #empty dictionary

        # Below we are calculating frequency of words in the corpus and storing it in word_doc in
```

```
 the form of dictionary
        for word in sorted_words:
            count = 0
            for sent in self.corpus:
                if word in sent:
                    count += 1
                else :
                    continue
            self.count_list.append(count)
        self.word_doc = dict(zip(sorted_words,self.count_list ))

        # Iterating over word_doc dictionary to calculate IDF of words and storing IDF values in i
df_list
        for i,j in self.word_doc.items():
            idf_i = 1 + math.log((1+len(self.corpus))/(1+j))
            self.idf_list.append(idf_i)

        # Storing words and corresponding IDF values in idf_dict dictionary as Key-Value pairs
        self.idf_dict = dict(zip(sorted_words,self.idf_list ))
        return self.idf_dict

    def my_vocab(self,max_features = None):
        '''1. This method returns a vocabulary with Words,Index and corresponding IDF values in th
e form of dictionary.
        2. If max_features is not specified then it will return Words and corresponding IDF values
        with words sorted alphabetically. Else it will return top (max_features) features based on
their idf value.
        3. This method is just to look at the vocabulary.
        '''
        final_vocab = {} #empty dictionary
        if max_features is None:
            idf_dict = self.my_idf_() #Calling my_idf_() method as it is
        else:
            idf_dict = self.my_idf_()
            t = dict(sorted(idf_dict.items(),key=operator.itemgetter(1),reverse=True)) #Sorting th
e output of my_idf_() in decreasing order of IDF values
            t_list = list(t.items())[:max_features] #Selecting top (max_features)
            idf_dict = dict(t_list)
        vocab_dict = {j:i for i,j in enumerate(idf_dict)} #This is an intermediary dictionary whic
h has words and their index as Key-Value pair
        for key in vocab_dict.keys(): #This will make sure to iterate over common keys in vocab_di
ct and idf_dict to merge values of both the dictionaries
            final_vocab[key] = [vocab_dict[key], idf_dict[key]] #Storing final output in final_voc
ab dictionary which has word as key and [index,IDF value] as value
        return final_vocab

    def my_transform(self,max_features = None):
        '''1. This method calculated TfIdf and returns Normalized Sparse Matrix
        2. If max_features is not specified then it will use all the words and corresponding IDF v
alues
        with words sorted alphabetically. Else it will use top (max_features) features based on th
eir idf value.
        '''
        # Below if-else statement is written to incorporate task-2 functionality of top 50 words.
        if max_features is None:
            idf_dict = self.my_idf_()
        else:
            idf_dict = self.my_idf_()
            t = dict(sorted(idf_dict.items(),key=operator.itemgetter(1),reverse=True))
```

```
            t_list = list(t.items())[:max_features]
            idf_dict = dict(t_list)
        self.tf_list = []
        self.tf_dict = {}

        # Calculating tf_idf value using formula provided in sklearn TfidfVectorizer
        for sent in (self.corpus):
            self.my_dict = dict()
            a = dict(Counter(sent.split())) # Storing words and their respective frequencies in a
 single document of the corpus as a dictionary key-value pair
            for i,j in a.items(): # Iterating over above dictionary with checks that length of wor
d should be greater than 1 and word should be in idf_dict that we have created
                if (len(i) > 1) & (i in idf_dict):
                    tf_i = j/len(sent.split(' ')) # Calculating Term Frequency here
                    idf_i = idf_dict[i] # Selecting correspnding IDF value of the word
                    tf_idf_i = tf_i*idf_i # Calculating tf_idf
                    self.my_dict.update({i:tf_idf_i}) # Storing word and corresponding IDF value i
n a dictionary
                else :
                    continue
            self.tf_dict.update({self.corpus.index(sent):self.my_dict}) # Storing index of the doc
ument in the corpus and words in that document with their tf_idf values as nested dictionaries
        my_dict = self.tf_dict
        vocab = {j:i for i,j in enumerate(idf_dict)}

        # Now we are converting calulated tf_idf values with document and words which is in nested
dictionary form to rows, columns and values in order to convert them to Sparse Matrix
        self.rows = []
        self.columns = []
        self.values = []
        for index,value in tqdm(my_dict.items()): #Iterating over my_dict where index is document
 index and value is a dictionary of words in that document with tf_idf values
            for word,tfidf in value.items(): # Iterating over value
                col_index = vocab.get(word,-1) # If word is not in the vocab then take -1 else ind
ex of the word
                self.rows.append(index) # This will have index of the document in the corpus
                self.columns.append(col_index) # Index of the words in the vocab
                self.values.append(tfidf) # Tf_IDF value

        # Here we are converting above rows, columns and values in Sparse matrix. Code is taken fr
om the reference provided in the Bog of Words
        matrix = csr_matrix((self.values, (self.rows,self.columns)), shape=(len(self.corpus),len(v
ocab)))
        matrix_norm = normalize(matrix,norm = 'l2') #Normalizing above matrix to achieve sklearn o
utput
        return matrix_norm
```

My Outputs for Task-1

In [30]:
```
my_vectorizer = myTfidf()
my_vectorizer.my_fit(corpus)
my_output = my_vectorizer.my_transform()
```

```
100%|██████████| 4/4 [00:00<?, ?it/s]
```

In [31]:
```
print (my_vectorizer.my_get_feature_names())
```

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

In [32]:
```
print(my_vectorizer.my_idf_())
```

```
{'and': 1.916290731874155, 'document': 1.2231435513142097, 'first': 1.5108256237659907, 'is': 1.0, 'one': 1.916290731874155, 'second': 1.9
16290731874155, 'the': 1.0, 'third': 1.916290731874155, 'this': 1.0}
```

In [33]:
```
print(my_output.shape)
```

```
(4, 9)
```

In [34]:
```
print(my_output[0])
```

```
  (0, 1)        0.4697913855799205
  (0, 2)        0.580285823684436
  (0, 3)        0.3840852409148149
  (0, 6)        0.3840852409148149
  (0, 8)        0.3840852409148149
```

In [35]:
```
print (my_output[0].toarray())
```

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

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 vocab. And make sure to print idf values for each term in your vocab.
  3. 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
  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.

In [3]:
```
# 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))
print (corpus[0])
```

```
Number of documents in corpus =  746
slow moving aimless movie distressed drifting young man
```

My Output for Task 2

In [8]:
```
## Please note that for this task I have already made required changes in the class:myTfidf.
## Method my_transform takes max_features as value as an input parameter to select top 50 features
based on their idfs.

my_vectorizer = myTfidf()
my_vectorizer.my_fit(corpus)
my_output = my_vectorizer.my_transform(50)
```

```
100%|████████████████████████████████████████████████████████| 741/741 [00:00<00:00, 374139.79it/s]
```

In [162]:
```
print (my_vectorizer.my_vocab(50))
```

In [10]:
```
print(my_output.shape)
```

```
(746, 50)
```

In [12]:
```
print (my_output[0])
```

```
  (0, 24)	1.0
```

In [14]:
```
print (my_output[0].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.
  1. 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.]]
```

In [16]:
```
print (my_output[0].toarray().shape)
```

```
(1, 50)
```