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.

</font>

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> </font>

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 [5]:

```
## 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 [6]:

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

In [7]:

```
# 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 [8]:

```
# 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 [9]:

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

skl_output.shape
```

Out[9]:

```
(4, 9)
```

In [10]:

```
# 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 [11]:

```
# 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 printi
ng 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 [12]:

```
# 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
```

In [13]:

```
import math
def sort_dictionary_by_value(dictionary, t_reverse=False):
    '''
    This function reverse the dictionary according to the value in ascending or descending order
    this function returns the sorted dictionary
    >>> temp_dict = {'a':20,'b':3,'c':5,'d':8}
    >>> sorted_dict = sort_dictionary_by_value(temp_dict,True)
    >>> sorted_dict
    O[1]: {'a':20,'d':8,'c':5,'b':3}
    '''
    # change the dictionary into list of tuples of dictionary's values and keys
    # step1-> access dictionary's values
    # step2-> change step1 to list
    # step3-> access dictionary's keys
    # step4-> change step 3 to list
    # step5-> zip step2 and step4(values and keys lists)
    # step6-> typecast it to list again
    temp_list = list(zip(list(dictionary.values()),list(dictionary.keys())))

    # Now, let's sort the list
    temp_list.sort(reverse = t_reverse)

    for i, item in enumerate(temp_list):
        temp = list(item)
        temp.reverse()
        temp = tuple(temp)
        # override the temp_list
        temp_list[i] = temp


    again_dict = dict(temp_list)

    return again_dict# returning the dictionary
```

In [14]:

```
# find how many review contain given word
def containing_word(dataset,word):

    count = 0
    for review in dataset:
        if word in review:
            count+=1
    return count
```

In [18]:

```
# let's create tfidf fit function

# let's do some change in fit function.
# impelement maximum feature functionality
def fit(dataset):
    '''
    It returns the dictionary of feature names and their idf
    '''

    # initialize an empty set to store unique words
    # in the corpus to find idf of that word
    unique_words = set()
    # initialize feature_idf dictionary to return the final result
    feature_idf = dict()

    # check if its list type or not
    if isinstance(dataset,(list,)):
        # first find the unique words in the dataset
        for row in dataset:
            for word in (row.split()):
                unique_words.add(word)

        # how many reviews are there in the dataset
        # to find the idf value
        no_of_reviews = len(dataset)


        for word in unique_words:# for each unique word in the reivew
                if len(word)<2:# It is found that adjective has no less than 2 words
                    continue

                # otherwise

                # find how many reviews containing the given 'word'
                reviews_contain_word = containing_word(dataset,word)

                # calculate the idf value according to the formula in sklearn official documentation
                # to overcome the problem of zero division error
                idf_value = 1+ math.log((1+no_of_reviews)/(1+reviews_contain_word))

                # storing the value in the dictionary
                # key: 'word'
                # value: 'idf' of that word
                feature_idf[word]=idf_value

        # sort dictionary by_value_return top 50 features
        sorted_feature_idf = sort_dictionary_by_value(feature_idf, True)

        #Now, sort according to their keys
        new_feature_idf = dict(sorted(sorted_feature_idf.items(),key = lambda kv:(kv[0], kv[1])))

        return new_feature_idf # returning unique words and their idf
    else:
        # if the dataset is not in the list format
        print('you need to pass list of sentence')
```

In [19]:

```
 # fitting the corpus to custom implementation of tfidfvectorizer
features = fit(corpus)
```

In [20]:

```
# print the features after fitting the corpus to the custom tfidfVectorizer
print('custom features')
print(list(features.keys()))
# features are as same as sklearn's .get_feature_names() returns
print('sklearn features')
print(vectorizer.get_feature_names())
```

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

In [21]:

```
# import numpy
# because sklearn's idf_ is in numpy.ndarray format
import numpy as np

# print idf of the gained features from the given corpus
print('custom idfs:')
print(np.array(list(features.values())))

#sklearn idfs
print('sklearn idfs:')
print(vectorizer.idf_)
```

```
custom idfs:
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
sklearn idfs:
[1.91629073 1.22314355 1.51082562 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]
```

In [22]:

```
# making transform function
def transform(dataset, vocab):
    # initializing lists of rows,columns,values to make sparse matrix
    rows = []
    columns = []
    values = []


    if  isinstance(dataset, (list,)):# if the dataset given is list

        for idx, row in enumerate(tqdm(dataset)):# for each review in the dataset

            word_freq = dict(Counter(row.split()))# find word frequency

            for word in (row.split()): # for each word in the reveiw
                if len(word)<2:
                    continue


                #find the column index from vocab
                idf = vocab.get(word)
                #find tf
                tf = word_freq[word]/len(row)
                #calculate tf_idf
                tf_idf = tf*idf

                #find the column index from vocab.features
                col_index = list(vocab.keys()).index(word)
                #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 tf-idf of the word
                values.append(tf_idf)

        # let's normalize our matrix
        tfidf_matrix = csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab)))
        normalized_matrix = normalize(tfidf_matrix)
        return normalized_matrix
    else:    # else print the error message
        print("you need to pass the dataset in list format")
```

In [28]:
```
tfidf = transform(corpus,features)
print('custom output:\n',tfidf[0])
# print('shape:',tfidf.toarray().shape)
```

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

custom output:
    (0, 1)	0.46979138557992045
  (0, 2)	0.5802858236844359
  (0, 3)	0.3840852409148149
  (0, 6)	0.3840852409148149
  (0, 8)	0.3840852409148149
```

In [29]:
```
print(list(features.keys()),'\n')
print('custom output',tfidf[0].toarray())
print('scikit-learn output:',skl_output[0].toarray())#sklern result
```

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

custom output [[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
scikit-learn output: [[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

custom output and sklearn output is same above

In [30]:
```
# shape of idf array
print('shape of idf array')
print(tfidf.toarray().shape)
```

```
shape of idf array
(4, 9)
```

In [ ]:

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 [41]:

```
# 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 [42]:

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

let's make fit and transform method

In [101]:

```
import math
def sort_dictionary_by_value(dictionary, t_reverse=False):
    '''
    This function reverse the dictionary according to the value in ascending or descending order
    this function returns the sorted dictionary
    >>> temp_dict = {'a':20,'b':3,'c':5,'d':8}
    >>> sorted_dict = sort_dictionary_by_value(temp_dict,True)
    >>> sorted_dict
    O[1]: {'a':20,'d':8,'c':5,'b':3}
    '''
    # change the dictionary into list of tuples of dictionary's values and keys
    # step1-> access dictionary's values
    # step2-> change step1 to list
    # step3-> access dictionary's keys
    # step4-> change step 3 to list
    # step5-> zip step2 and step4(values and keys lists)
    # step6-> typecast it to list again
    temp_list = list(zip(list(dictionary.values()),list(dictionary.keys())))

    # Now, let's sort the list
    temp_list.sort(reverse = t_reverse)

    for i, item in enumerate(temp_list):
        temp = list(item)
        temp.reverse()
        temp = tuple(temp)
        # override the temp_list
        temp_list[i] = temp

    # selecting top 50 features according to their idf values
    t_list = temp_list[0:50]

    again_dict = dict(t_list)

    return again_dict# returning the dictionary
```

In [ ]:

In [102]:

```
# let's create tfidf fit function

# let's do some change in fit function.
# impelement maximum feature functionality
def fit(dataset):
    '''
    It returns the dictionary of feature names and their idf
    '''

    # initialize an empty set to store unique words
    # in the corpus to find idf of that word
    unique_words = set()
    # initialize feature_idf dictionary to return the final result
    feature_idf = dict()

    # check if its list type or not
    if isinstance(dataset,(list,)):
        # first find the unique words in the dataset
        for row in dataset:
            for word in (row.split()):
                unique_words.add(word)

        # find how many reviews are there in the dataset. it will help to find the idf value
        no_of_reviews = len(dataset)


        for word in unique_words:# for each unique word in the reivew
                if len(word)<2:# It is found that adjective has no less than 2 words
                    continue

                # otherwise

                # find how many reviews containing the given 'word'
                reviews_contain_word = containing_word(dataset,word)

                # calculate the idf value according to the formula in sklearn official documentation
                # to overcome the problem of zero division error
                idf_value = 1+ math.log((1+no_of_reviews)/(1+reviews_contain_word))

                # storing the value in the dictionary
                # key: 'word'
                # value: 'idf' of that word
                feature_idf[word]=idf_value

        # sort dictionary by_value_return top 50 features
        sorted_feature_idf = sort_dictionary_by_value(feature_idf, True)
        #Now, sort according to their keys
        new_feature_idf = dict(sorted(sorted_feature_idf.items(),key = lambda kv:(kv[0], kv[1])))

        return new_feature_idf # returning unique words and their idf
    else:
        # if the dataset is not in the list format
        print('you need to pass list of sentence')
```

In [117]:

```
# making transform function
def transform(dataset, vocab):
    # initializing lists of rows,columns,values to make sparse matrix
    rows = []
    columns = []
    values = []


    if  isinstance(dataset, (list,)):# if the dataset given is list

        for idx, row in enumerate(tqdm(dataset)):# for each review in the dataset

            word_freq = dict(Counter(row.split()))# find word frequency

            for word in (row.split()): # for each word in the reveiw
                if len(word)<2:
                    continue

                # if word is not present in our top features then skip that word
                if word not in list(vocab.keys()):
                    continue
                else:

                    #find the column index from vocab
                    idf = vocab.get(word,0)
                    #find tf
                    tf = word_freq[word]/len(row)
                    #calculate tf_idf
                    tf_idf = tf*float(idf)

                    #find the column index from vocab.features

                    col_index = list(vocab.keys()).index(word)
                    #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 tf-idf of the word
                    values.append(tf_idf)

        # let's normalize our matrix
        tfidf_matrix = csr_matrix((values,(rows,columns)),shape=(len(dataset),len(vocab)))
        normalized_matrix = normalize(tfidf_matrix)
        return normalized_matrix
    else:    # else print the error message
        print("you need to pass the dataset in list format")
```

In [118]:

```
# custom implementation
# fit the corpus to custom fit method
task2_fit = fit(corpus)
```

In [119]:

```
# sklearn implementation
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=50)
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

In [120]:
```
# custom features
print('custom top 50 features')
print(list(task2_fit.keys()))

# sklearn features
print('sklearn top 50 features')
print(vectorizer.get_feature_names())
```

```
custom top 50 features
['waster', 'wasting', 'wave', 'waylaid', 'wayne', 'weaker', 'weariness', 'weaving', 'website', 'wedding', 'we
ight', 'welsh', 'went', 'whenever', 'whine', 'whites', 'whoever', 'wide', 'widmark', 'wife', 'wih', 'wild',
'william', 'willie', 'wily', 'within', 'witticisms', 'woa', 'wondered', 'wong', 'wont', 'worked', 'worry', 'w
orthless', 'worthwhile', 'wouldnt', 'woven', 'wow', 'wrap', 'writers', 'wrote', 'yardley', 'yawn', 'yelps',
'younger', 'youthful', 'youtube', 'yun', 'zillion', 'zombiez']
sklearn top 50 features
['acting', 'actors', 'also', 'bad', 'best', 'better', 'cast', 'character', 'characters', 'could', 'even', 'ev
er', 'every', 'excellent', 'film', 'films', 'funny', 'good', 'great', 'like', 'little', 'look', 'love', 'mad
e', 'make', 'movie', 'movies', 'much', 'never', 'no', 'not', 'one', 'plot', 'real', 'really', 'scenes', 'scri
pt', 'see', 'seen', 'show', 'story', 'think', 'time', 'watch', 'watching', 'way', 'well', 'wonderful', 'wor
k', 'would']
```

In [121]:
```
vectorizer.idf_
```

Out[121]:
```
array([3.97847903, 4.67162621, 4.39718936, 3.62708114, 4.57154275,
       4.78285184, 4.67162621, 4.57154275, 4.15032928, 4.39718936,
       4.03254625, 4.48057097, 4.84347646, 4.97700786, 2.7718781 ,
       4.67162621, 4.78285184, 3.78742379, 4.18207798, 4.00514727,
       4.72569343, 4.62033291, 4.57154275, 4.48057097, 4.67162621,
       2.71822539, 4.48057097, 4.72569343, 4.72569343, 4.35796865,
       2.89756631, 3.57301392, 4.35796865, 4.57154275, 4.08970466,
       4.78285184, 4.67162621, 4.03254625, 4.48057097, 4.78285184,
       4.57154275, 4.67162621, 3.95250354, 4.72569343, 4.67162621,
       4.52502273, 4.11955762, 4.67162621, 4.67162621, 4.28386067])
```

```
In [122]: print('custom idfs:\n',np.array(list(task2_fit.values())))
```

```
custom idfs:
 [6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918 6.922918
 6.922918 6.922918]
```

implement transform

```
In [123]: custom_result = transform(corpus,task2_fit)
```

```
100%|██████████████████████████████████████████████████████████████████████████████| 746/746 [00:00<00:00, 350
62.20it/s]
```

```
In [124]: custom_result.toarray()
```

```
Out[124]: 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.]])
```

In [126]:
```
skl_output.toarray()
```

Out[126]:
```
array([[0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.52029442, 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       ...,
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ]])
```

I have used idf value to select maximum features(50).

Here the result is different because sklearn doesn't use idf to select top features.

sklearn uses tf(term frequency) to select top features