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]:

```
# 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 [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.        ]
```

In [5]:

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

skl_output.shape
```

Out[5]:

```
(4, 9)
```

In [6]:

```
# 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 [7]:

```
# 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 [8]:

```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in c:\users\hp\anaconda3\lib\site-packages (4.48.2)
```

```
mysql-connector-python 8.0.21 requires protobuf>=3.0.0, which is not installed.
distributed 1.21.8 requires msgpack, which is not installed.
You are using pip version 10.0.1, however version 20.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Code to get unique BoW for a corpus

In [9]:

```
# Write your code here.
# Make sure its well documented and readable 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

def fit(corpus):
    """
    This function takes a dataset of sentences and returns the list of unique words in that datase
t/ corpus
    """
    # Check if the dataset is a list
    if isinstance(corpus, (list,)):
        # Initialize the unique words with an empty set
        unique_words = set()
        # Check every row
        for sentence in corpus:
            for word in sentence.split(" "):
                # Add only those words having more than 2 latters
                if len(word) < 2:
                    continue;
                unique_words.add(word)
        #Convert the set to list for sorting
        unique_words = sorted(list(unique_words))
        # Create a dictionary of unique words and its coresponding dimension value
        vocab = {i:j for j,i in enumerate(unique_words)}

        return vocab
    else:
        print('The corpus should be a list of sentences')
```

Calculate IDF for words in BoW for a corpus.

In [10]:

```
def getIDF(vocabList, corpus):
    """
    This function takes all unique words in a corpus and the corpus itself to generate IDF values
for each word
    """
    # Initialize IDF dict {word : IDF value}
    idfs = dict()
    # Traverse through each word
    for word in vocabList:

        # numerator is the number of documents in the corpus
        numerator = 1 + len(corpus)
        # List of sentences containing the word
        sentencesWithWord = [sentence for sentence in corpus if word in sentence]
        # denominator is the number of sentences containing the word
        denominator = 1 + len(sentencesWithWord)

        # Calculate IDF for the word
        idf = round(1 + math.log(numerator / denominator), 8)
        idfs.update({word: idf})

    return idfs
```

Get FT-IDF sparse matrix

In [21]:

```
def transform(corpus, vocab, idfs):
    """
    this function takes the corpus, vocab words, idf values and calculates TF-IDF value for each s
entence and word
    """
    # Initialize rows, columns and values to generate sparse matrix
    rows = []
    columns = []
    values = []

    if isinstance(corpus, (list,)):
        # Take each sentence from corpus
```

```
        for rowIdx, sentence in enumerate(tqdm(corpus)):
            # Make a list of words from the sentence (String)
            sentence = sentence.split(" ")
            # Get unique words
            uniqueWords = set(sentence)
            # Get the list of unique words (We are making a list again to use count() method to
get frequency of a word)
            words = list(uniqueWords)
            for word in words:
                # We have already discarded the words with length less than 2 from BoW
                # Word must be present in the BoW and words with IDF
                if word in list(vocab.keys()):
                    # Calculate term frequency
                    tf = sentence.count(word) / len(sentence)
                    # Calculate tf-idf value
                    tf_idf = tf * idfs.get(word)

                    # Build sparse matrix with row id, column id and value
                    rows.append(rowIdx)
                    columns.append(vocab.get(word))
                    values.append(tf_idf)

        sparse_matrix = csr_matrix((values, (rows, columns)), shape=(len(corpus), len(vocab)))
        return normalize(sparse_matrix, norm='l2')
    else:
        print("Corpus should be a lst of sentences")
```

In [22]:

```
#Call fit and transform

vocab = fit(corpus)
print(vocab)

idfs = getIDF(list(vocab.keys()), corpus)
print(idfs)

transformedMatrix = transform(corpus = corpus, vocab = vocab, idfs = idfs)
print(transformedMatrix.toarray())
```

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

```
{'and': 0, 'document': 1, 'first': 2, 'is': 3, 'one': 4, 'second': 5, 'the': 6, 'third': 7,
'this': 8}
{'and': 1.91629073, 'document': 1.22314355, 'first': 1.51082562, 'is': 1.0, 'one': 1.91629073, 'se
cond': 1.91629073, 'the': 1.0, 'third': 1.91629073, 'this': 1.0}
[[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]]
```

Compare results for FTIDFvectoriser and Custom_FTIDFvectoriser

In [23]:

```
# Check both results

print("Vocab\n")
print(vectorizer.get_feature_names())
print("\n")
print(list(vocab.keys()))
print("\n")
print("IDFs\n")
print(list(idfs.values()))
print("\n")
print(vectorizer.idf_)
print("\n")
print("Vectorized sparse matrix\n")
```

```
print("Vectorized sparse matrix\n")
print(skl_output.toarray())
print("\n")
print(transformedMatrix.toarray())
```

```
Vocab

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


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


IDFs

[1.91629073, 1.22314355, 1.51082562, 1.0, 1.91629073, 1.91629073, 1.0, 1.91629073, 1.0]


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


Vectorized sparse matrix

[[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]]


[[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]]
```

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 [24]:

```python
# 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
```

Updated fit method to generate vocab with limited terms/ words as 50 with highest IDF values

In [25]:

```python
import numpy as np

def fit(corpus, top):
    """
    This function takes a dataset of sentences and returns the list of unique words with top IDF scores in that dataset/ corpus
    """
    # Check if the dataset is a list
    if isinstance(corpus, (list,)):
        # Initialize the unique words with an empty set
        unique_words = set()
        # Check every row
        for sentence in corpus:
            for word in sentence.split(" "):
                # Add only those words having more than 2 latters
                if len(word) < 2:
                    continue;
                unique_words.add(word)

        #Convert the set to list for sorting
        unique_words = sorted(list(unique_words))

        # get only words having top IDF values
        # np array of the unique words
        np_unique_words = np.array(unique_words)
        # get the IDF values
        idfs = np.array(list(getIDF(unique_words, corpus).values()))
        # Get indexes of words with top idf values
        topIdx = np.argsort(-1 *idfs)[:top]
        # get the list of words after filteration
        unique_words_top = sorted(np_unique_words[topIdx].tolist())
        # get top idfs
        topIdfs = idfs[topIdx].tolist()
        # generate dict of term and its idf
        topIdfTerms = dict(zip(unique_words_top, topIdfs))
        # Sort the dict by terms
        topIdfTerms = dict(sorted(topIdfTerms.items()))
        # generate vocab from the unique words
        vocab = {j:i for i,j in enumerate(unique_words_top)}
        return vocab, topIdfTerms
    else:
        print('The corpus should be a list of sentences')
```

Call fit and transform method with top 25 IDF values

In [26]:

```python
vocab, idfs = fit(corpus, 25)

print("Vocab\n")
print(list(vocab.keys()))
```

```
print("\n")
print("IDFs\n")
print(idfs)

transfomedMatrix = transform(corpus = corpus, vocab = vocab, idfs = idfs)

print("\n")
print(transfomedMatrix.shape)
print(len(corpus))
print(transfomedMatrix.toarray())
```

```
100%|█████████████████████████████████████████████████████████████████████████████| 746/746
[00:00<00:00, 41289.93it/s]
```

```
Vocab

['aailiyah', 'microsoft', 'middle', 'mighty', 'mindblowing', 'miner', 'minor', 'mirrormask', 'mise
rable', 'miserably', 'misplace', 'mistakes', 'modest', 'mollusk', 'momentum', 'monica', 'monolog',
'monotonous', 'monster', 'monstrous', 'monumental', 'morons', 'motivations', 'mountain', 'moved']


IDFs

{'aailiyah': 6.922918, 'microsoft': 6.922918, 'middle': 6.922918, 'mighty': 6.922918,
'mindblowing': 6.922918, 'miner': 6.922918, 'minor': 6.922918, 'mirrormask': 6.922918,
'miserable': 6.922918, 'miserably': 6.922918, 'misplace': 6.922918, 'mistakes': 6.922918,
'modest': 6.922918, 'mollusk': 6.922918, 'momentum': 6.922918, 'monica': 6.922918, 'monolog': 6.92
2918, 'monotonous': 6.922918, 'monster': 6.922918, 'monstrous': 6.922918, 'monumental': 6.922918,
'morons': 6.922918, 'motivations': 6.922918, 'mountain': 6.922918, 'moved': 6.922918}


(746, 25)
746
[[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.]]
```