Problem statement

- Task 1

Build a TFIDF Vectorizer from scratch & compare its results with Sklearn:

In [1]:

```
#IMPORT REQUIRED LIBRARIES
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 [2]:

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

In [3]:

```
#FIT FUNCTION

def fit(corpus):
#STORING UNIQUE WORDS TO DICTIONARY WITH THEIR RESPECTIVE INDEX
    #initialize a empty set
    unique_words=set()
    #check whether given corpus in list format or not
    if isinstance(corpus,(list,)):
        #iterate over every row in the corpus list
        for idx,row in enumerate(corpus):
            #for every word in the row
            for word in row.split():
                #skipping the words with lenth 1
                if len(word)<2:
                    continue
                #adding each word to the set
                unique_words.add(word)
        #converting set to list and then sort it
        unique_words=sorted(list(unique_words))
        #storing every word to the dictionary with word as the key and their respective index as value
        vocab={j:i for i,j in enumerate(unique_words)}

#TO CALCULATE TERM FREQUENCY OF EVERY WORD
        tf=[]
        #iterate over evey row in the corpus
        for row in corpus:
            temp=[]
            #for evey key in the vocab
            for key in vocab:
                no_of_times=0
                #to count number of times the word(key=each unique word) in the dictionary appeared in that ROW
                for word in row.split():
                    if word==key:
                        no_of_times+=1
                #calculate TF by finding number of times each unique word appeared in a row divided by total number of words in that row
                temp.append(no_of_times/len(row.split()))
            #appending TF values of all words in each row
            tf.append(temp)


#CALCULATE INVERSE DOCUMENTRY FREQUENCY OF EACH WORD
        idf=[]
        #for evey key in the vocab
        for key in vocab:
            no_of_doc=0
            #iterate over evey row in the corpus
            for row in corpus:
                #to count number of times the word(key=each unique word) in the dictionary appeared in the DOCUMENT
                for word in row.split():
                    if word==key:
                        no_of_doc+=1
                        #break helps to move to next row(document) if a word in a row is considered once.
                        break
                    #if word is not match with key,just continue with next word
                    else:
                        continue
            #calculate IDF and append it on list
            idf.append(1+(math.log(((1+len(corpus))/(1+no_of_doc)))))


#CALCULATE NORMALISED VALUES OF TF_IDF VALUES
        tf_idf=[]
        #for every row(tf values of all words in that perticular row) in tf list
        for lst in tf:
            temp=[]
            #multipy TF values of each word with its IDF value respectively
            for idx in range(len(lst)):
                temp.append(lst[idx]*idf[idx])
            tf_idf.append(temp)
        #calculate normalization of TF_IDF values
        tf_idf_norm=normalize(tf_idf, norm='l2')
        return vocab,tf_idf_norm,idf

    else:
        print("you need to pass list of sentence")
```

In [4]:

```
#Calling fit function
vocab,tf_idf_norm,idf=fit(corpus)
print('VOCAB IS\n',vocab)
print('*'*120)
#print('NORMALISED TF_IDF VALUES ARE\n',tf_idf_norm)
#print('*'*120)
print('IDF VALUES ARE\n',idf)
```

```
VOCAB IS
 {'and': 0, 'document': 1, 'first': 2, 'is': 3, 'one': 4, 'second': 5, 'the': 6, 'third': 7, 'this': 8}
************************************************************************************************************************
IDF VALUES ARE
 [1.916290731874155, 1.2231435513142097, 1.5108256237659907, 1.0, 1.916290731874155, 1.916290731874155, 1.0, 1.916290731874
155, 1.0]
```

In [5]:

```
#TRANSFORM FUNCTION TO CREATE FEATURE MATRIX

def transform(corpus,vocab):
    rows = []           #list to store row index of the word
    columns = []        #list to store column index of the word
    values = []         #list to store normalised tf_idf values of words
    #check whether given corpus in list format or not
    if isinstance(corpus,(list,)):
        #iterate over every row in the corpus list
        for idx,row in enumerate(tqdm(corpus)):
            #for every word in the row
            for word in row.split():
                #get column index of a word if it present in vocab(dictionary) or else set it to -1
                col_index=vocab.get(word,-1)
                #to check if column index of a word if it present in vocab(dictionary) is not -1
                if col_index!=-1:
                    #append row index of the word
                    rows.append(idx)
                    #append column index of the word
                    columns.append(col_index)
                    #append normalised tf_idf of that word
                    values.append(tf_idf_norm[idx][col_index])
        #creating sparse matrix from feature matrix
        return csr_matrix((values,(rows,columns)),shape=(len(corpus),len(vocab)))
```

In [6]:

```
#Calling transform function
matrix=transform(corpus,vocab)
print('MATRIX SHAPE IS\n',matrix.shape)
print('*'*100)
print('FIRST ROW OF SPARSE MATRIX\n',matrix[0])
print('*'*100)
print('FIRST ROW(DOCUMENT) OF SPARSE MATRIX IS CONVERTED INTO DENSE MATRIX FORMAT\n',matrix[0].toarray())
```

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

MATRIX SHAPE IS
 (4, 9)
****************************************************************************************************
FIRST ROW OF SPARSE MATRIX
   (0, 1)       0.4697913855799205
  (0, 2)        0.580285823684436
  (0, 3)        0.3840852409148149
  (0, 6)        0.3840852409148149
  (0, 8)        0.3840852409148149
****************************************************************************************************
FIRST ROW(DOCUMENT) OF SPARSE MATRIX IS CONVERTED INTO DENSE MATRIX FORMAT
 [[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Result are matching when comparing with sklearn tf_idf vectoriser output

In [7]:

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

In [8]:

```
print(vectorizer.get_feature_names())
```

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

C:\Users\natar\anaconda3\envs\tf_gpu\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function get_feature
_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out i
nstead.
  warnings.warn(msg, category=FutureWarning)
```

In [9]:

```
print(vectorizer.idf_)
```

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

In [10]:

```
skl_output.shape
```

Out[10]:

```
(4, 9)
```

In [11]:

```
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 [12]:

```
print(skl_output[0].toarray())
```

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

TASK 2

Implement max features functionality:

In [14]:

```
# 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 [15]:

```
#FIT FUNCTION

def fit(corpus):
    #initialize a empty set
    unique_words=set()
    #check whether given corpus in list format or not
    if isinstance(corpus,(list,)):
        #iterate over every row in the corpus list
        for idx,row in enumerate(corpus):
            #for every word in the row
            for word in row.split():
                #skipping the words with lenth less than 1
                if len(word)<2:
                    continue
                 #adding each word to the set
                unique_words.add(word)
        #converting set to list and then sort it
        unique_words=sorted(list(unique_words))
        #storing every word to the dictionary with word as the key and their respective index as value
        vocab={j:i for i,j in enumerate(unique_words)}


 #CALCULATE INVERSE DOCUMENTRY FREQUENCY OF TOP 50 IDF WORDS
        idf=[]                          #list to store idf values
        vocab_idf={}                    #dictionary to store words with their respective idf values
        sorted_vocab_idf={}             #dictionary to store top 50 idf values as values and respective words as keys
        sorted_vocab_index={}           #dictionary to store new index of top 50 idf values as values and respective words as keys
        #for evey key(word) in the vocab
        for key in vocab:
            no_of_doc=0
            #iterate over evey row in the corpus
            for row in corpus:
                #to count number of times the word(key=each unique word) in the dictionary appeared in the DOCUMENT
                for word in row.split():
                    if word==key:
                        no_of_doc+=1
                        #break helps to move to next row(document) if a word in a row is considered once.
                        break
                    #if word is not match with key,just continue with next word
                    else:
                        continue
            #calculate IDF and append it on list
            idf.append(1+(math.log(((1+len(corpus))/(1+no_of_doc)))))
            #dictionary to store word as a key and its idf values as value
            vocab_idf[key]=(1+(math.log(((1+len(corpus))/(1+no_of_doc)))))
        #vocab_idf.items()------ retuns values and key pairs
        #item[1]----- in lamda function returns key values and then key values are sorted in desending order
        #[:50]------ slicing is used to get top 50 key vales of sorted idf values
        #sorted_vocab_idf----dictionary containing top 50 idf values as values and respective words as keys
        sorted_vocab_idf={k: v for k, v in sorted(vocab_idf.items(), key=lambda item: item[1],reverse=True)[:50]}
        #give new index to sorted keys(words)
        s=list(sorted_vocab_idf.keys())
        sorted_vocab_index={j:i for i,j in enumerate(s)}


 #TO CALCULATE TERM FREQUENCY OF EVERY WORD
        tf=[]
        #iterate over evey row in the corpus
        for row in corpus:
            temp=[]
            #for evey key in the sorted_vocab_idf
            for key in sorted_vocab_idf:
                no_of_times=0
                #to count number of times the word(key=each unique word) in the dictionary appeared in that ROW
                for word in row.split():
                    if word==key:
                        no_of_times+=1
                #calculate TF by finding number of times each unique word appeared in a row divided by total number of words in that row
                temp.append(no_of_times/len(row.split()))
            #appending TF values of all words in each row
            tf.append(temp)

 #CALCULATE NORMALISED VALUES OF TF_IDF VALUES
        tf_idf=[]
        #for every row(tf values of all words in that perticular row) in tf list
        for lst in tf:
            temp=[]
            #multipy TF values of each word with its IDF value respectively
            for idx in range(len(lst)):
                temp.append(lst[idx]*idf[idx])
            tf_idf.append(temp)
        #calculate normalization of TF_IDF values
        tf_idf_norm=normalize(tf_idf, norm='l2')
        return sorted_vocab_index,tf_idf_norm,sorted_vocab_idf


    else:
```

```
        print("you need to pass list of sentence")
```

In [16]:

```
sorted_vocab_index,tf_idf_norm,sorted_vocab_idf=fit(corpus)
print('IDF VALUES ARE\n',sorted_vocab_idf)
```

```
IDF VALUES ARE
 {'aailiyah': 6.922918004572872, 'abandoned': 6.922918004572872, 'abroad': 6.922918004572872, 'abstruse': 6.92291800457287
2, 'academy': 6.922918004572872, 'accents': 6.922918004572872, 'accessible': 6.922918004572872, 'acclaimed': 6.922918004572
872, 'accolades': 6.922918004572872, 'accurate': 6.922918004572872, 'accurately': 6.922918004572872, 'achille': 6.922918004
572872, 'ackerman': 6.922918004572872, 'actions': 6.922918004572872, 'adams': 6.922918004572872, 'add': 6.922918004572872,
'added': 6.922918004572872, 'admins': 6.922918004572872, 'admiration': 6.922918004572872, 'admitted': 6.922918004572872, 'a
drift': 6.922918004572872, 'adventure': 6.922918004572872, 'aesthetically': 6.922918004572872, 'affected': 6.92291800457287
2, 'affleck': 6.922918004572872, 'afternoon': 6.922918004572872, 'aged': 6.922918004572872, 'ages': 6.922918004572872, 'agr
ee': 6.922918004572872, 'agreed': 6.922918004572872, 'aimless': 6.922918004572872, 'aired': 6.922918004572872, 'akasha': 6.
922918004572872, 'akin': 6.922918004572872, 'alert': 6.922918004572872, 'alike': 6.922918004572872, 'allison': 6.9229180045
72872, 'allow': 6.922918004572872, 'allowing': 6.922918004572872, 'alongside': 6.922918004572872, 'amateurish': 6.922918004
572872, 'amaze': 6.922918004572872, 'amazed': 6.922918004572872, 'amazingly': 6.922918004572872, 'amusing': 6.9229180045728
72, 'amust': 6.922918004572872, 'anatomist': 6.922918004572872, 'angel': 6.922918004572872, 'angela': 6.922918004572872, 'a
ngelina': 6.922918004572872}
```

In [17]:

```
#TRANSFORM FUNCTION TO CREATE FEATURE MATRIX

def transform(corpus,sorted_vocab_index):
    rows = []
    columns = []
    values = []
    #check whether given corpus in list format or not
    if isinstance(corpus,(list,)):
        #iterate over every row in the corpus list
        for idx,row in enumerate(tqdm(corpus)):
            #for every word in the row
            for word in row.split():
                #get column index of a word if it present in vocab(dictionary) or else set it to -1
                col_index=sorted_vocab_index.get(word,-1)
                #to check if column index of a word if it present in vocab(dictionary) is not -1
                if col_index!=-1:
                    #append row index of the word
                    rows.append(idx)
                    #append column index of the word
                    columns.append(col_index)
                    #append normalised tf_idf of that word
                    values.append(tf_idf_norm[idx][col_index])
        #creating sparse matrix from feature matrix
        return csr_matrix((values,(rows,columns)),shape=(len(corpus),len(sorted_vocab_idf)))
```

In [18]:

```
#calling transform function
matrix=transform(corpus,sorted_vocab_index)
print('MATRIX SHAPE IS\n',matrix.shape)
print('*'*100)
print('FIRST ROW OF SPARSE MATRIX\n',matrix[0])
print('*'*100)
print('FIRST ROW(DOCUMENT) OF SPARSE MATRIX IS CONVERTED INTO DENSE MATRIX FORMAT\n',matrix[0].toarray())
```

```
100%|██████████████████████████████████████████████████████████████████████████| 746/746 [00:00<00:00, 247759.19it/s]

MATRIX SHAPE IS
 (746, 50)
****************************************************************************************************
FIRST ROW OF SPARSE MATRIX
   (0, 30)      1.0
****************************************************************************************************
FIRST ROW(DOCUMENT) OF SPARSE MATRIX IS CONVERTED INTO DENSE MATRIX FORMAT
 [[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. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0.]]
```

SKLEARN TF_IDF VECTORIZER

In [19]:

```
#SKLEARN TF_IDF VECTORIZER
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=50) #max_features for tern frequency only not for idf..... #max_featuresint, default=None
                                              #If not None, build a vocabulary that only consider the top max_features ordered by term f
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

In [20]:

```
print(vectorizer.get_feature_names())
```

```
['acting', 'actors', 'also', 'bad', 'best', 'better', 'cast', 'character', 'characters', 'could', 'even', 'ever', 'every',
'excellent', 'film', 'films', 'funny', 'good', 'great', 'like', 'little', 'look', 'love', 'made', 'make', 'movie', 'movie
s', 'much', 'never', 'no', 'not', 'one', 'plot', 'real', 'really', 'scenes', 'script', 'see', 'seen', 'show', 'story', 'thi
nk', 'time', 'watch', 'watching', 'way', 'well', 'wonderful', 'work', 'would']

C:\Users\natar\anaconda3\envs\tf_gpu\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function get_feature
_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out i
nstead.
  warnings.warn(msg, category=FutureWarning)
```

In [21]:

```
print(vectorizer.idf_)
```

```
[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 [22]:

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

Out[22]:

```
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., 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.]])
```

In [23]:

```
skl_output.shape
```

Out[23]:

```
(746, 50)
```