TF-IDF Vectorization

> Instructions Using either Python or R for this assignment, program your own function to take a set of text files and perform TF-IDF vectorization on them.
>
> Your work can be programmed in the .R, .rmd, .py, or .ipynb format.
>
> Please compare the similarity of the files as a result of this.

Strategy

1. Load all text documents and vectorize == aggregate as one *Master list* of words
2. For each document: Count how often the words in the master list appear == Term Frequency (**TF**) vectors for each document
3. For each word: Calculate the Inverse Document Frequency (**IDF**) to create an IDF vector as follows:

   > **log(# of total documents) / [log(# documents using specific word +1)**

File similarity from these vectors:

- Goal of IDF: if a word appears in almost every document (eg."the", "and") the IDF evaluates to ~0.
- Dot product of TF (per document) and IDF (weights of word importance): outputs a scalar & is based on how closely the two align.
- These output vectors can be analyzed with clustering algorithms such as k-means to estimate how similar they are to one another.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
```

Example

I will start by importing one text file and demonstrating the process that the function later will perform on all text files

Step 1: Import text file and create master list of unique words that appear

In [2]:
```
master_list = []
with open("C:\\Users\\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\\Alice_in_Wonder
    for line in Alice:
        words = line.split()
        for word in range(0,len(words)):
            if words[word] not in master_list:
                master_list.append(words[word])
```

In [3]:
```
#view the master list as needed
#master_list
```

In [4]:
```
#Checking the master list before proceeding
unique_check = np.unique(master_list, return_counts=True) #return counts parameter = T
print(unique_check)

unique_count = 0
not_unique = 0

for i in range(0,len(unique_check[1])): #index second position for the unique counts r
    if unique_check[1][i] == 1:
        unique_count += 1
    if unique_check[1][i] != 1:
        not_unique += 1

print(f'Number of unique entries in the master list: {unique_count}')
print(f'Number of DUPLICATE entries in the master list: {not_unique}')
```

```
(array(['(Alice', '(And,', '(As', ..., '“‘We', '“‘—found', '“’Tis'],
      dtype='<U46'), array([1, 1, 1, ..., 1, 1, 1], dtype=int64))
Number of unique entries in the master list: 5167
Number of DUPLICATE entries in the master list: 0
```

Step 2: Term-Frequency Vector

In [5]:
```
# Use the master list to create Term-Frequency (TF) Vector

tf_Alice = np.zeros(shape=(len(master_list)),dtype=np.int32) #create placeholder array

with open("C:\\Users\\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\\Alice_in_Wonder
    for term in range(0,len(master_list)):    #loop through the master list
        Alice.seek(0)                         #return to the beginning of the document
        for line in Alice:                    #loop through each line in the document (
            words = line.split()
            for word in range(0,len(words)): #loop through each word in the document
                if words[word] == master_list[term]:
                    tf_Alice[term] += 1       #count each instance of the word, in the
```

In [6]:
```
print(f'Length of TF vector for Alice in Wonderland: {len(tf_Alice)}')
print(f'Length of (example) master list {len(master_list)}')
print('')
print(f'The first 10 values in the TF vector for Alice in Wonderland are: \n{tf_Alice[
print(f'The corresponding words from the master list are: \n{master_list[0:10]}')
```

```
Length of TF vector for Alice in Wonderland: 5167
Length of (example) master list 5167

The first 10 values in the TF vector for Alice in Wonderland are:
[  12    1    1 1513    1  221  332   11  706   43]
The corresponding words from the master list are:
['CHAPTER', 'I.', 'Down', 'the', 'Rabbit-Hole', 'Alice', 'was', 'beginning', 'to', 'g
et']
```

IDF Vector, dot product & Clustering

Calculated later when we have several documents to compare to each other.

Function

In [7]:

```
#function will accept list of filenames
file_Alice = "C:\\Users\\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\\Alice_in_Wor
file_Two_Cities = "C:\\Users\\Cal\Desktop\\DSCI 508 Machine Learning\\Week 7\\A_Tale_c
file_Frankenstein = "C:\\Users\\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\\Frank
file_Great_Gatsby = "C:\\Users\\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\\The_Gr
file_Pride_and_Prejudice = "C:\\Users\Cal\\Desktop\\DSCI 508 Machine Learning\\Week 7\
```

In [8]:

```
master_list = []

#this function combines Step 1 and Step 2 above, and then also creates an IDF vector f
def tf_idf_vector_func(files_vector): #accepts a list of filenames
    hold_tf_vectors = []
    hold_idf_vector = []
    hold_dot_products = []
    #Create master list of all words that appear in all documents (Similar to step 1 a
    for filename in range(0,len(files_vector)):
        with open(files_vector[filename],'r', encoding='utf-8') as file:
            for line in file:
                words = line.split()
                for word in range(0,len(words)):
                    if words[word] not in master_list:
                        master_list.append(words[word])
    #Create tf vectors now that the master list is created (Similar to step 2 above)
    for filename in range(0,len(files_vector)):
        hold_tf_vectors.append(np.zeros(shape=(len(master_list)),dtype=np.int32)) #hol
        with open(files_vector[filename],'r', encoding='utf-8') as file:
            for term in range(0,len(master_list)):   #Loop through the master list
                file.seek(0)                          #return to the beginning of the d
                for line in file:                     #Loop through each line in the dc
                    words = line.split()
                    for word in range(0,len(words)): #Loop through each word in the dc
                            if words[word] == master_list[term]:
                                hold_tf_vectors[filename][term] += 1  #count each
    #Create vector for IDF
    for word in range(0,len(master_list)):            #IDF vectors are per word, loop t
        word_count = 0                                #Count how often each word appear
        for vector in range(0,len(hold_tf_vectors)): #By looping over the tf vectors
            if hold_tf_vectors[vector][word] > 0:
                word_count +=1
        idf_value = np.log(len(files_vector))/np.log(word_count+1)  #IDF = (# total dc
        hold_idf_vector.append(idf_value)             #store IDF value in vector, same
    #Dot products of TF and IDF vectors
```

```
    for tf_vector in range(0,len(hold_tf_vectors)):
        dot_product = np.dot(hold_tf_vectors[tf_vector],hold_idf_vector)
        hold_dot_products.append(dot_product)
    return hold_tf_vectors, hold_idf_vector, hold_dot_products
```

In [9]:
```
vectorize_books = tf_idf_vector_func([file_Alice, file_Two_Cities, file_Frankenstein,
```

Examine resulting vectors

In [10]:
```
#idf vector
print(f'The shape of the IDF vector is: {np.shape(vectorize_books[1])}')
print(f'The shape of the master list is: {np.shape(master_list)}')
print('')
#print(f'The IDF vector is: \n {vectorize_books[1]} ')
```

```
The shape of the IDF vector is: (37357,)
The shape of the master list is: (37357,)
```

As expected, the IDF array holds only a single vector which has a value for each word index. These numbers have a lot of floating point decimals as the result of division of logarithms.

In [11]:
```
#tf vectors
tf_Alice = vectorize_books[0][0]
tf_Two_Cities = vectorize_books[0][1]
tf_Frankenstein = vectorize_books[0][2]
tf_Great_Gatsby = vectorize_books[0][3]
tf_Pride_and_Prejudice = vectorize_books[0][4]

print(f'The first five words of the master list are: \n {master_list[0:5]}')
tf_array = [tf_Alice,tf_Two_Cities,tf_Frankenstein,tf_Great_Gatsby,tf_Pride_and_Prejuc

for i in range(0,len(tf_array)):
    print(np.shape(tf_array[i]))
    print(tf_array[i][0:5])
```

```
The first five words of the master list are:
 ['CHAPTER', 'I.', 'Down', 'the', 'Rabbit-Hole']
(37357,)
[  12    1    1 1513    1]
(37357,)
[  44   10    0 7347    0]
(37357,)
[   0    1    0 3897    0]
(37357,)
[   0    0    0 2201    0]
(37357,)
[  59    0    0 4069    0]
```

As expected, all are the same length as the master list, and are whole numbers since they are integer counts of the words.

And also as expected, "the" is a very common word!

In [12]:
```
#dot products
#vectorize_books[2]
```

```
for dots in range(0,len(vectorize_books[2])):
    print(vectorize_books[2][dots])
```

```
29327.867030090445
152801.7373287466
81668.23575407494
54957.31596921146
132854.09055423026
```

The dot product of two vectors outputs a scalar. **I wonder if element-wise matrix multiplcation is actually more practical since it outputs a vector?**

In [13]:
```
#element-wise multiplication rather than dot product

hadamard_products = []

for tf_vector in range(0,len(tf_array)):
    hadamard_prod = np.multiply(tf_array[tf_vector], vectorize_books[1])
    print(hadamard_prod[0:5])
    hadamard_products.append(hadamard_prod)
```

```
[1.39315686e+01 1.16096405e+00 2.32192809e+00 1.35904378e+03
 2.32192809e+00]
[  51.08241809   11.60964047    0.         6599.40161932    0.        ]
[0.00000000e+00 1.16096405e+00 0.00000000e+00 3.50045843e+03
 0.00000000e+00]
[   0.           0.           0.         1977.03592815    0.        ]
[  68.4968788    0.           0.         3654.95647053    0.        ]
```

Since these outputs are vectors, this is more suited to a clustering algorithm than scalars are.

Perform clustering on vectors & estimate similarity

In [14]:
```
km_classifier = KMeans(n_clusters=2, random_state=0, n_init=10)
k_means_books = km_classifier.fit(X=hadamard_products)

print(f' The cluster assignment is as follows: {k_means_books.labels_}')

#View clusters if desired (very high dimensionality)
#k_means_books.cluster_centers_
```

```
 The cluster assignment is as follows: [0 1 0 0 1]
C:\Users\Cal\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1436: UserWarnin
g: KMeans is known to have a memory leak on Windows with MKL, when there are less chu
nks than available threads. You can avoid it by setting the environment variable OMP_
NUM_THREADS=1.
  warnings.warn(
```

The assigned clusters are [0 1 0 0 1]. From this method, this conveys that books 1,3 and 4 are similar, while books 2, and 5 are similar.

Books 1,3,4 are:

- Alice in Wonderland
- Frankenstein

- The Great Gatsby

Books 2 and 5 are:

- A Tale of Two Cities
- Pride and Prejudice

I think it's understandable that A Tale of Two Cities and Pride and Prejudice are rather alike!