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.

<base>

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 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.

- TFIDF Example: We multiply term frequency times inverse document frequency: $TFIDF(t,d,D) = TF(t,d) \times IDF(t,D)$

There are many variations on how to calculate term frequency and inverse document frequency. Some common variations include using a boolean frequency of 1 if the term appears in the document, or using the logarithm of the term frequency plus 1 in place of the term frequency itself, or dividing the term frequency by the total number of terms in the document, or using a logarithmic version of inverse document frequency.

For our purposes, we'll define term frequency as the raw count of a term in a document, i.e. the number of times that term t occurs in document d. Then we define inverse document frequency idf(t, D) as follows:

$$idf(t, D) = \log \frac{\text{Total number of documents}}{\text{number of documents with term } t \text{ in it}}$$

Intuitively, this means that if a term t occurs in many documents, its inverse document frequency will be low, because it suggests that t is a common term and we don't learn much about any particular document from seeing a document containing t. The idea is that we want to weigh down the frequent terms while scaling up the rare ones. Computing the TFIDF:

$$TFIDF(t, d, D) = \underbrace{\log(1 + \text{raw count of term } t \text{ in } d)}_{\text{term frequency}} \times \underbrace{\log \frac{\text{Total number of documents}}{\text{number of documents with term } t \text{ in it}}}_{\text{inverse document frequency}}$$

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 $\log(10{,}000{,}000 / 1{,}000) = 9$. Thus, the TF-idf weight is the product of these quantities: $0.03 * 9 = 0.27$.

Task-1

1. Build a TFIDF Vectorizer & compute for 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 TFIDF vectorizer with the outcomes of that with of sklearn implementation 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 a list of English stopwords, which it does not use by default. 2) Sklearn TfidfVectorizer applies some default regex to select words/tokens, for example, it drops all single character words. 3) Sklearn TfidfVectorizer applies L2-norm to the output matrix. 4) Sklearn applies smoothing to the idf values. 5) Sklearn applies L2-norm to the tfidf vectors. 6) Sklearn TfidfVectorizer has many other parameters which gives you more flexibility in the tokenization step, such as, you can ignore words that occur in more than 90% of the documents, or you can ignore words that occur in less than 5 documents, or you can specify to use only top 5000 words ordered by their term frequency across the corpus, etc. 7) The default output of sklearn tfidf vectorizer is a sparse matrix.

- Steps to approach this task:
  1. You may choose to write both fit and transform methods for your custom implementation of tfidf vectorizer.
  2. Print out the alphabetically sorted vocab 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 tfidf values from your implementation and check if its the same as that of sklearn tfidf vectorizer tfidf values.
  4. Keep in mind that sklearn uses a slightly different version of tfidf than what we have discussed in class, for more details refer to the sklearn documentation.
  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/preprocessing.html#normalization
  6. After completing the above steps, print the shape of your custom implementation and compare it with sklearn implementation of tfidf vectorizer.
  7. Finally, for 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 sklearn 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 of fit and of sklearn implementation would match 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 refer to the 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 you try to print things which are not part of this task.

Corpus

```
# Declaration Collection of string documents

corpus = [
    'This is the first document.',
    'This is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
```

SkLearn Implementation

```
from sklearn.feature_extraction.text import TfidfVectorizer,CountVectorizer
from sklearn.preprocessing import normalize
vectorizer= TfidfVectorizer()
sk_output = vectorizer.transform(corpus)

# 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']

# 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 1.91629073 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]

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

sk_output.shape

(4, 9)

# sklearn tfidf values for first line of the above corpus.
# Here the output is a sparse matrix

print(sk_output[0])

  (0, 3)	0.3779644730092272
  (0, 8)	0.5804860367950862
  (0, 6)	0.3779644730092272
  (0, 4)	0.6804615491833403
  (0, 1)	0.4736296010332684
  (0, 0)	0.2995784974171539

# 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(sk_output[0].toarray())

[[0.         0.47363960 0.         0.37796447 0.         0.
  0.         0.58048604]]
```

Your custom implementation

```
# write your code here.
# Try not to hardcode any values.
# 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 math import log10
from sklearn.preprocessing import normalize
import math ^
import operator
from sklearn.preprocessing import normalize
import numpy as np

# Helper(s) for TF values for unique words
def DefCountUniqueWord(cur_corpus):
    IdfDict = {}
    for word in uniqueWordList:
        wordCount = 0
        for sentence in cur_corpus:
            IdfDict[word] = 0 + math.log(1 + len(corpus)) / (1 + wordCount)
    return IdfDict

# This function is used to compute the TF of document contains the word in term
def InverseTermFrequency(word,word_corpus):
    count = 0
    for document in corpus:
        if word in document:
            count += 1
        else:
            pass
    return count

def Frequency(document):
    frequency = {}
    documentList = document.split(' ')

    for word in documentList:
        frequency[word] = frequency.get(word, 0) + 1

    return frequency

def fit(corpus):
    unique_word = set()
    for document in corpus:
        for word in document.split(' '):
            if len(word) >= 2:
                unique_word.add(word)
    uniqueWordList = list(unique_word)
    IdfDict = DefCreateWordCorpusCount(corpus)
    # Storing the unique words in the corpus in the form of a dictionary
    vocabulary = {word: index for index,word in enumerate(uniqueWordList)}

    return (vocabulary,IdfDict)

corpus = [
    'This is the first document.',
    'This is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
vocabulary,IdfDict = fit(corpus)
print("Vocabulary: {0}".format(vocabulary))

print("Idf values: {0}".format(IdfDict))

Vocabulary: {'and': 1, 'document': 2, 'first': 3, 'is': 4, 'one': 5, 'second': 6, 'the': 7, 'third': 8, 'this': 9}
Idf values: {'and': 1.9162907318741551, 'document': 1.2231435513142097, 'first': 1.9162907318741551, 'is': 1.9162907318741551, 'one': 1.9162907318741551, 'second': 1.9162907318741551, 'the': 1.9162907318741551, 'third': 1.9162907318741551, 'this': 1.9162907318741551}

# TDIF for all the unique features
print("Inverse Document Frequency of unique Features : {0}".format(np.array(list(IdfDict.value())))

Inverse Document Frequency of unique Features : [1.91629073 1.22314355 1.91629073 1.         1.91629073 1.91629073
 1.         1.91629073 1.        ]

def transform(corpus,vocabulary,IdfDict):
    rows,columns = len(corpus),len(vocabulary)
    uniqueWordList = list(vocabulary.keys())

    for index, document in enumerate(corpus):
        word_frequency = Frequency(document)
        for word,count in word_frequency.items():
            if word in vocabulary:
                column_index = vocabulary.get(word,None)
                if column_index != None:
                    rows.append(index)
                    columns.append(column_index)
                    # Compute the frequency of the given word
                    termFrequency = count / len(document.split(' '))
                    inverseTermFrequency = IdfDict[word]
                    TF_IDF = termFrequency * inverseTermFrequency[word]
                    values.append(TF_IDF)

    return normalize(csr_matrix((values, (rows,columns)), shape=(len(corpus),len(vocabulary)),norm = 'l2',axis=1, copy=True, return_norm=False)

tfidf_transformedMatrix = transform(corpus,vocabulary,IdfDict)
print(type(tfidf_transformedMatrix))

<class 'scipy.sparse.csr.csr_matrix'>

print("Shape of TFIDF Vectorizer: {0}".format(tfidf_transformedMatrix.shape))

Shape of TFIDF Vectorizer: (4, 9)

print(tfidf_transformedMatrix[0])

  (0, 3)	0.4697413865706369
  (0, 8)	0.5804860367950862
  (0, 6)	0.3779644730092272
  (0, 4)	0.6804615491833403
  (0, 1)	0.4736296010332684
  (0, 0)	0.2995784974171539

print("Sparsity Matrix of the corpus 1:{0}".format(tfidf_transformedMatrix[0].toarray()))

Sparsity Matrix of the corpus 1:[[0.         0.47363960 0.         0.37796447 0.         0.
  0.         0.58048604]]
```

Task-2

2. Implement more 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 tfidf 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 tfidf 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 given a little bit more freedom, this time **round_vocabs**. 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 tfidf values and print out the words in the sorted vocab after you fit your data. Here you should be getting only 50 terms in your vocab. And make sure to print tfidf values for each term in your vocab.
  3. Make sure the output of your custom 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/preprocessing.html#normalization
  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.

```
# Below is the code to load the cleaned_strings pickle file provided
# Here corpus is of list type

import pickle
corpus = open("cleaned_strings", 'rb')[ as f
    corpus = pickle.load(f)

# printing the length of the corpus loaded
print("Number of documents in corpus = {0}".format(len(corpus)))

Number of documents in corpus = 746

vectorizer = TfidfVectorizer(lf)(corpus)

print(vectorizer.idf_)

5996

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

def DefCreateWordCorpusCount(corpus):
    IdfDict = {}
    for word in uniqueWordList:
        wordCount = 0
        for sentence in corpus:
            IdfDict[word] = 0 + math.log(1 + len(corpus)) / (1 + wordCount)
    return IdfDict

def InverseTermFrequency(word,word_corpus):
    count = 0
    for document in corpus:
        if word in document:
            count += 1
        else:
            pass
    return count
def Frequency(document):
    frequency = {}
    documentList = document.split(' ')

    for word in documentList:
        frequency[word] = frequency.get(word, 0) + 1

    return frequency

def fit(corpus):
    unique_word = set()
    for document in corpus:
        for word in document.split(' '):
            if len(word) >= 2:
                unique_word.add(word)
    uniqueWordList = list(unique_word)
    IdfDict = DefCreateWordCorpusCount(corpus)
    # Storing the unique words in the corpus in the form of a dictionary
    vocabulary = {word: index for index,word in enumerate(maximumIdfFeaturesDictionary)}

    maximumIdfFeaturesDictionary = dict(list(maximumIdfFeaturesDictionary.items())[:50])

    maximumIdfFeaturesDictionary = list(maximumIdfFeaturesDictionary.keys())

    maximumIdfFeaturesDictionary.sort()

    vocabulary = {word: index for index,word in enumerate(maximumIdfFeaturesDictionary)}

    return (vocabulary,maximumIdfFeaturesDictionary)

vocabulary,maximumIdfFeaturesDictionary = fit(corpus)
print("Vocabulary: {0}".format(vocabulary))
print("\n\n")

print("Top IDFvalues: {0}".format(maximumIdfFeaturesDictionary))

Vocabulary: {'a': 0, 'aegean': 1, 'a. ''delta' 2, 'borders': 3, 'b.''aqean': 4, 'carry': 5, 'chase': 6, 'chosen': 7, 'commander': 8, 'crisis': 9, 'cultivation': 10, 'daughters': 11, 'drawn': 12, 'entertained': 13, 'expansion': 14, 'exploits': 15, 'followed': 16, 'flamboyantly': 17, 'forces': 18, 'gaith': 19, 'gaul': 20, 'greatness': 21, 'hadrian': 22, 'hadrian''s': 23, 'hadrian''s': 24, 'hadrian''s': 25, 'hadrian''s': 26, 'hadrian': 27, 'hadrian': 28, 'hadrian': 29, 'hadrian': 30, 'hadrian': 31, 'hadrian': 32, 'hadrian': 33, 'hadrian': 34, 'hadrian': 35, 'hadrian': 36, 'hadrian': 37, 'hadrian': 38, 'hadrian': 39, 'hadrian': 40, 'hadrian': 41, 'hadrian': 42, 'hadrian': 43, 'hadrian''s': 44, 'hadrian': 45, 'hadrian': 46, 'hadrian': 47, 'hadrian': 48, 'hadrian': 49}

Top IDFvalues: [0.9162907318741551, 0.9162907318741551, 0.9162907318741551, 
```