TF-IDF Instructions

January 29, 2021

1 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.

1.1 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:

Sklearn has its vocabulary generated from idf sroted in alphabetical order

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.

$IDF(t) = 1 + \log_e \frac{1 + \text{Total number of documents in collection}}{1 + \text{Number of documents with term t in it}}$.

Sklearn applies L2-normalization on its output matrix.

The final output of sklearn tfidf vectorizer is a sparse matrix.

```
<br>
<li>Steps to approach this task:
<ol>
    <li> You would have to write both fit and transform methods for your custom implementation c
    <li> Print out the alphabetically sorted voacb after you fit your data and check if its the
    <li> Print out the idf values from your implementation and check if its the same as that of
    <li> Once you get your voacb and idf values to be same as that of sklearns implementation of
    <li> Make sure the output of your implementation is a sparse matrix. Before generating the f
    <li> After completing the above steps, print the output of your custom implementation and co
    <li> To check the output of a single document in your collection of documents,  you can conv
    </ol>
</li>
<br>
```

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.

1.1.1 Corpus

[1]:
```
corpus_1 = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]
```

1.1.2 SkLearn Implementation

Using sklearn to get the tfidf values and then implementing it myself to compare the values

[2]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus_1)
skl_output = vectorizer.transform(corpus_1)
```

[3]:
```
print(vectorizer.get_feature_names())
```

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

[4]:
```
print(vectorizer.vocabulary_)
```

```
{'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5, 'and': 0,
'third': 7, 'one': 4}
```

[5]:
```
print(vectorizer.idf_)
```

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

[6]:
```
skl_output.shape
```

[6]:
```
(4, 9)
```

[7]:
```
print(skl_output[0])
```

```
  (0, 8)        0.38408524091481483
  (0, 6)        0.38408524091481483
  (0, 3)        0.38408524091481483
  (0, 2)        0.5802858236844359
  (0, 1)        0.46979138557992045
```

[8]:
```
skl_output.toarray()
```

[8]:
```
array([[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]])
```

[9]:
```
print(skl_output[0].toarray())
```

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

1.1.3 Your custom implementation

Importing all necessary modules and packages

[10]:
```
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
```

[11]:
```
def fit_1(dataset):
    """
    fit is a function that takes the dataset and returns the vocabulary which is␣
 ↪a dataset with words
    as keys and their dimensions as values
    """
    unique_words = set()
    if isinstance(dataset, (list,)):
        for row in dataset:
            for word in row.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)
        unique_words = sorted(list(unique_words))
        vocab = {j:i for i,j in enumerate(unique_words)}
        return vocab
    else:
        print("you need to pass list of sentance")
```

[12]:
```
def transform_1(dataset,vocab):
    """
```

```
    transform is a function that takes the dataset(corpus) and vocab as inputs␣
↪and returns the tfidf csr matrix.
    """
    # Calculating the word repetition matrix. The row denotes the sentence and␣
↪the column denotes the word
    # who's dimension is taken from the vocab.
    # First the words_reps is calculated as a csr matrix and then, it is␣
↪converted to a normal array.

    rows = []
    columns = []
    values = []
    for idx, row in enumerate(tqdm(dataset)):
        word_freq = dict(Counter(row.split()))
        for word, freq in word_freq.items():
            if len(word) < 2:
                continue
            col_index = vocab.get(word, -1)
            if col_index !=-1:
                rows.append(idx)
                columns.append(col_index)
                values.append(freq)
    word_reps = csr_matrix((values, (rows,columns)),␣
↪shape=(len(dataset),len(vocab))).toarray()

    # Calculating the tf of the dataset. tf can be calculated by dividing each␣
↪row of the word repition matrix
    # with the length of the sentence/document.

    tf = []
    for x_row, doc in enumerate(corpus_1):
        doc_words = list(doc.split())
        doc_len = 0
        for word in doc_words:
            if len(word) >= 2:
                doc_len += 1
        tf.append(word_reps[x_row]/doc_len)

    # Calculating the IDF values. This can be calculated using word reps matrix.␣
↪We iterate over all rows of each
    # column while keeping track of non zero values and at the end we calculated␣
↪the IDF values. These values are
    # stored in a list where each column
    # corresponds to it's respective word's dimension.

    idf = []
```

```
    temp = 0
    n_docs_1 = len(dataset)
    for n_col in range(len(word_reps[0])):
        for n_row in range(len(dataset)):
            if word_reps[n_row][n_col] > 0:
                temp += 1
        idf.append(1+math.log((1+n_docs_1)/(1+temp)))
        temp = 0

    # Calculating the tf-idf values in a csr matrix. If either the tf or idf␣
↪value of a word in a sentence is a zero,
    # it's skipped. If it's a non zero then the row value, column value, tf-idf␣
↪value are added to the lists and
    # csr matrix is returned.

    rows = []
    columns = []
    values = []
    for row in range(len(word_reps)):
        for col in range(len(word_reps[0])):
            if tf[row][col] != 0 and idf[col] != 0:
                rows.append(row)
                columns.append(col)
                values.append(tf[row][col]*idf[col])
    return csr_matrix((values, (rows,columns)), shape=(len(dataset),len(vocab)))
```

Getting the vocabulary using the corpus

[13]:
```
vocab_1 = fit_1(corpus_1)
```

Calculating the tf-idf and normalizing to l2. Also printing the output obtained using sklearn for comparision.

[14]:
```
tf_idf_1 = normalize(transform_1(corpus_1, vocab_1).toarray(), norm='l2')
print(tf_idf_1)
print("\n\n")
print(skl_output.toarray())
```

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

[[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]]
```

1.2 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:

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.

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.

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

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.

```
</li>
<br>
```

[15]:
```
def fit_2(dataset):
    """
    The fit function for this task is unchanged. It still returns the same␣
 ↪vocabulary. The idf selection is
    performed in the transform function
```

```
    """
    unique_words = set()
    if isinstance(dataset, (list,)):
        for row in dataset:
            for word in row.split(" "):
                if len(word) < 2:
                    continue
                unique_words.add(word)
        unique_words = sorted(list(unique_words))
        vocab = {j:i for i,j in enumerate(unique_words)}

        return vocab
    else:
        print("you need to pass list of sentance")
```

[16]:
```
def transform_2(dataset,vocab):
    """
    The transform function is modified in such a way that it calculated the idf␣
↪values, sorted and the top values
    and their corresponding indices are kept track of.
    Then these indices are used to select the correspond tf-values from the␣
↪entire tf value set.
    The transform function still returns the same value i.e., the tf-idf values.␣
↪But this time it only returns
    tf-idf for 50 features
    """
    # Calculating the word repetition matrix. The row denotes the sentence and␣
↪the column denotes the word
    # who's dimension is taken from the vocab.
    # First the words_reps is calculated as a csr matrix and then, it is␣
↪converted to a normal array.
    rows = []
    columns = []
    values = []
    for idx, row in enumerate(dataset):
        word_freq = dict(Counter(row.split()))
        for word, freq in word_freq.items():
            if len(word) < 2:
                continue
            col_index = vocab.get(word, -1)
            if col_index !=-1:
                rows.append(idx)
                columns.append(col_index)
                values.append(freq)
    word_reps = csr_matrix((values, (rows,columns)),␣
↪shape=(len(dataset),len(vocab))).toarray()
```

```
   # Calculating the tf of the dataset. tf can be calculated by dividing each␣
↪row of the word repition
   # matrix with the length of the sentence/document.
   tf = []
   for x_row, doc in enumerate(dataset):
       doc_words = list(doc.split())
       doc_len = len(doc_words)
       tf.append(word_reps[x_row]/doc_len)

   # Calculating the idf value# Calculating the IDF values. This can be␣
↪calculated using word reps matrix.
   # We iterate over all rows of each column while keeping track of non zero␣
↪values and at the end we calculated
   # the IDF values. These values are stored in a list where each column
   # corresponds to it's respective word's dimension.

   idf = []
   temp = 0
   n_docs_1 = len(dataset)
   for n_col in range(len(word_reps[0])):
       for n_row in range(len(dataset)):
           if word_reps[n_row][n_col] > 0:
               temp += 1
       idf.append(1+math.log((1+n_docs_1)/(1+temp)))
       temp = 0

   # Obtaining the top 50 values' indices by converting the converting into an␣
↪np array, argsorting and getting
   #the last 50 values. These indices are stored in top_idf_indices list
   # Then the top idf values are obtained using list comprehension
   # The top tf values can be obtained by deleting all columns except the␣
↪top_idf_indices
   # all_other_indices will contain indices of top 51-2886 values. This is␣
↪necessary to drop those columns from
   #the tf-matrix
   # tf_top will contain the tf values for the correspoinding top idf values.

   top_idf_indices = numpy.array(idf).argsort()[-50:][::-1]
   top_idf_indices.sort()
   all_other_indices = []
   for i in range(len(word_reps[0])):
       if i not in top_idf_indices:
           all_other_indices.append(i)
   top_idf = [idf[i] for i in top_idf_indices]
   tf_np = numpy.array(tf)
   tf_top = numpy.delete(tf_np, all_other_indices, axis=1)
```

```
    tf_top = tf_top.tolist()

    # Printing top features and their IDF values
    top_features = {list(vocab.keys())[list(vocab.values()).index(i)]:idf[i] for␣
↪i in top_idf_indices}

    # Calculating the tf-idf values and returning their csr matrix
    rows = []
    columns = []
    values = []
    for row in range(len(word_reps)):
        for col in range(50):
            if tf_top[row][col] != 0 and idf[col] != 0:
                rows.append(row)
                columns.append(col)
                values.append(tf_top[row][col]*top_idf[col])
    return csr_matrix((values, (rows,columns)), shape=(len(dataset),50)),␣
↪top_features
```

Importing the dataset

[17]:
```
import pickle
with open('./datasets/cleaned_strings', 'rb') as f:
    corpus_2 = pickle.load(f)

print("Number of documents in corpus_2 = ",len(corpus_2))
```

```
Number of documents in corpus_2 =  746
```

Fitting the dataset to get the vocabulary

[18]:
```
vocab_2 = fit_2(corpus_2)
```

Passing the dataset and the vocabulary to the transform function to get the csr matrix, top features and then normalizing tfidf to l2

[19]:
```
tf_idf_2, top_features = transform_2(corpus_2, vocab_2)
tf_idf_2 = normalize(tf_idf_2, norm = 'l2')
```

Printing the top_features based on idf values

[20]:
```
print(top_features)
```

```
{'gone': 6.922918004572872, 'gosh': 6.922918004572872, 'goth':
6.922918004572872, 'gotta': 6.922918004572872, 'gotten': 6.922918004572872,
'government': 6.922918004572872, 'grade': 6.922918004572872, 'gradually':
6.922918004572872, 'grainy': 6.922918004572872, 'granted': 6.922918004572872,
'graphics': 6.922918004572872, 'grasp': 6.922918004572872, 'grates':
6.922918004572872, 'halfway': 6.922918004572872, 'ham': 6.922918004572872,
'handle': 6.922918004572872, 'handles': 6.922918004572872, 'hang':
```

```
6.922918004572872, 'hankies': 6.922918004572872, 'hanks': 6.922918004572872,
'happiness': 6.922918004572872, 'happy': 6.922918004572872, 'harris':
6.922918004572872, 'hatred': 6.922918004572872, 'havilland': 6.922918004572872,
'hay': 6.922918004572872, 'hayao': 6.922918004572872, 'hayworth':
6.922918004572872, 'hbo': 6.922918004572872, 'heads': 6.922918004572872,
'hearts': 6.922918004572872, 'heartwarming': 6.922918004572872, 'heche':
6.922918004572872, 'heels': 6.922918004572872, 'heist': 6.922918004572872,
'helen': 6.922918004572872, 'hellish': 6.922918004572872, 'helms':
6.922918004572872, 'help': 6.922918004572872, 'helping': 6.922918004572872,
'hendrikson': 6.922918004572872, 'hernandez': 6.922918004572872, 'hero':
6.922918004572872, 'heroes': 6.922918004572872, 'heroine': 6.922918004572872,
'heroism': 6.922918004572872, 'hes': 6.922918004572872, 'hide':
6.922918004572872, 'higher': 6.922918004572872, 'zombiez': 6.922918004572872}
```

Printing the normalized tf_idf values

[21]:
```
print(tf_idf_2)
```

```
  (19, 10)      0.7745966692414833
  (19, 12)      0.25819888974716115
  (19, 19)      0.25819888974716115
  (19, 32)      0.25819888974716115
  (19, 33)      0.25819888974716115
  (19, 37)      0.25819888974716115
  (19, 38)      0.25819888974716115
  (94, 1)       1.0
  (101, 0)      1.0
  (104, 15)     1.0
  (109, 36)     0.7071067811865476
  (109, 49)     0.7071067811865476
  (132, 43)     1.0
  (135, 4)      0.5773502691896258
  (135, 7)      0.5773502691896258
  (135, 20)     0.5773502691896258
  (180, 48)     1.0
  (191, 25)     0.7071067811865475
  (191, 44)     0.7071067811865475
  (197, 27)     1.0
  (222, 46)     1.0
  (225, 23)     1.0
  (232, 47)     1.0
  (234, 13)     1.0
  (236, 40)     1.0
  (253, 17)     1.0
  (270, 34)     1.0
  (277, 2)      1.0
  (323, 6)      1.0
  (343, 9)      1.0
  (371, 3)      1.0
```

```
(421, 45)     1.0
(430, 24)     1.0
(437, 42)     1.0
(459, 35)     1.0
(462, 39)     1.0
(475, 18)     1.0
(532, 16)     1.0
(533, 21)     1.0
(539, 14)     1.0
(572, 22)     1.0
(610, 8)      1.0
(625, 29)     1.0
(628, 41)     1.0
(633, 11)     1.0
(644, 31)     1.0
(660, 28)     1.0
(681, 30)     1.0
(703, 5)      1.0
(714, 26)     1.0
```