Assignment_3_Instructions

January 9, 2024

@meer_21

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.

`<li>`

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:

```
<li> As a part of this task you will be implementing TFIDF vectorizer on a collection of text c
<br>
<li> You should compare the results of your own implementation of TFIDF vectorizer with that o
<br>
<li> Sklearn does few more tweaks in the implementation of its version of TFIDF vectorizer, so
   <ol>
    <li> Sklearn has its vocabulary generated from idf sroted in alphabetical order</li>
    <li> Sklearn formula of idf is different from the standard textbook formula. Here the cons
```

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

```
    <li> Sklearn applies L2-normalization on its output matrix.</li>
    <li> The final output of sklearn tfidf vectorizer is a sparse matrix.</li>
</ol>
<br>
<li>Steps to approach this task:
<ol>
    <li> You would have to write both fit and transform methods for your custom implementation
    <li> Print out the alphabetically sorted voacb after you fit your data and check if its th
    <li> Print out the idf values from your implementation and check if its the same as that o
    <li> Once you get your voacb and idf values to be same as that of sklearns implementation
    <li> Make sure the output of your implementation is a sparse matrix. Before generating the
    <li> After completing the above steps, print the output of your custom implementation and
    <li> To check the output of a single document in your collection of documents,  you can co
    </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]:
```
## 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',
]
```

1.1.2 SkLearn Implementation

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

[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']
```

```
/home/ajaz/anaconda/anaconda3/lib/python3.9/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 instead.
  warnings.warn(msg, category=FutureWarning)
```

[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.        ]
```

[5]:
```
# shape of sklearn tfidf vectorizer output after applying transform method.

skl_output.shape
```

[5]:
```
(4, 9)
```

[6]:
```
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[0]
```

[8]:
```
<1x9 sparse matrix of type '<class 'numpy.float64'>'
        with 5 stored elements in Compressed Sparse Row format>
```

[9]:
```
# 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
```

[10]:
```
# 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]]
```

1.1.3 Your custom implementation

[11]:
```
# Write your code here.
# Make sure its well documented and readble 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
```

[12]:
```
def fit(dataset):
    """
    dataset: list of all sentences
    This function will take list of sentences and will return the corpus␣
 ↪basically all unique words in those sentences.
    """
    unique_words = set()
    vocab = dict()
    if isinstance(dataset, list): # only takes list of sentences so checking␣
 ↪that
        for sentence in dataset: # in each sentence
            for word in sentence.split(' '): # in each word
                if len(word) >= 2: # neglecting words less than 2 letters
                    unique_words.add(word)
        unique_words = sorted(list(unique_words))
    else:
        print('Please pass list of sentences')
    return unique_words

def count_sentences_having_word(dataset, target_word):
    """
    This function will take list of sentences and returns the count of target␣
 ↪word in whole list
    """
    count = 0
    for sentence in dataset: # looking in each sentence
        count_dict = Counter(sentence.split(' ')) # counting each words
        if count_dict.get(target_word): # if target word exists increment the␣
 ↪counter
            count += 1
    return count

def get_idf(dataset, vocab):
    """
    dataset: list of sentences
    vocab: corpus, all unique words
    This function takes list of sentences and returns idf values for each word.
```

```
    """
    if not isinstance(dataset, list):
        print('Please pass dataset as list of strings')
        return
    idf = dict()
    for word in vocab: # taking each word
        num = 1 + len(dataset) # Total number of sentences
        den = count_sentences_having_word(dataset, word) + 1 # Number of␣
 ↪senteces having the given word
        result = round(1 + math.log(num/den), 8) # idf value
        idf[word] = result
    return idf

def tf(sentence, word):
    """
    This function takes sentence and word and returns the term frequency of the␣
 ↪given word
    """
    word_split = Counter(sentence.split(" ")) # Counting each word
    if word_split.get(word):
        return word_split.get(word) / len(word_split) # number of occurance of␣
 ↪word divided by number of total words
    return 0

def transform(dataset, vocab):
    """
    dataset: list of sentences
    vocab: all unique words
    This function takes list of senteces and all unique words and returns a␣
 ↪sparse matrix having tfidf values for each word of the sentence
    """
    idf_values = get_idf(dataset, vocab) # get the idf value of all the unique␣
 ↪words
    tfidf = list()
    rows, col = list(), list() # These rows and columns are used to make sparse␣
 ↪matrix
    count_row = 0
    for sentence in dataset: # looking into all the sentences one by one.
        values = list()
        count_col = 0
        for word in set(sentence.split(' ')): # taking each word.
            tf_val = tf(sentence, word) # fetching tf of the given word.
            result = tf_val * idf_values.get(word, 0) # calculating the tfidf␣
 ↪value of the given word.
            values.append(result)
            col.append(count_col)
```

```
            rows.append(count_row)
            count_col += 1 # increasing the column count for each word
        count_row += 1  # increasing the row count for each sentence
        tfidf.append(normalize(numpy.array(values).reshape(1, -1))[0]) #␣
↪normalizing and then appending the value to tfidf list.
    tfidf = numpy.concatenate(tfidf, axis=None) # flattening the list to make␣
↪sparse matrix.
#     print(len(tfidf), len(rows), len(col))
    return csr_matrix((tfidf, (rows, col)), shape=(len(dataset), len(vocab)))
```

[13]:
```
vocab = fit(corpus)
print(vocab)
idf = get_idf(corpus, vocab)
# print(idf)
tfidf = transform(corpus, vocab)
# print(tfidf)
```

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

[14]:
```
print(tfidf)
```

```
  (0, 0)	0.5802858228626505
  (0, 1)	0.3840852413282814
  (0, 2)	0.3840852413282814
  (0, 3)	0.3840852413282814
  (0, 4)	0.46979138558088085
  (1, 0)	0.2810886742563164
  (1, 1)	0.2810886742563164
  (1, 2)	0.5386476207853688
  (1, 3)	0.2810886742563164
  (1, 4)	0.687623597789329
  (2, 0)	0.2671037878474866
  (2, 1)	0.2671037878474866
  (2, 2)	0.5118485126000253
  (2, 3)	0.5118485126000253
  (2, 4)	0.5118485126000253
  (2, 5)	0.2671037878474866
  (3, 0)	0.5802858228626505
  (3, 1)	0.3840852413282814
  (3, 2)	0.3840852413282814
  (3, 3)	0.3840852413282814
  (3, 4)	0.46979138558088085
```

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 vo

This task is similar to your previous task, just that here your vocabulary is limited to on

Here you will be give a pickle file, with file name cleaned_strings. You w

Steps to approach this task:

     You would have to write both fit and transform methods for your custom implementation
     Now sort your vocab based in descending order of idf values and print out the words in
     Make sure the output of your implementation is a sparse matrix. Before generating the
     Now check the output of a single document in your collection of documents, you can co

[15]:

```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
```

[16]:

```python
def fit(dataset):
    """
    dataset: list of all sentences
    This function will take list of sentences and will return the corpus basically all unique words in those sentences.
    """
    unique_words = set()
    vocab = dict()
    if isinstance(dataset, list): # only takes list of sentences so checking that
        for sentence in dataset: # in each sentence
            for word in sentence.split(' '): # in each word
                if len(word) >= 2: # neglecting words less than 2 letters
                    unique_words.add(word)
        unique_words = sorted(list(unique_words))
    else:
        print('Please pass list of sentences')
    return unique_words

def count_sentences_having_word(dataset, target_word):
```

```
    """
    This function will take list of sentences and returns the count of target␣
↪word in whole list
    """
    count = 0
    for sentence in dataset: # looking in each sentence
        count_dict = Counter(sentence.split(' ')) # counting each words
        if count_dict.get(target_word): # if target word exists increment the␣
↪counter
            count += 1
    return count

def get_idf(dataset, vocab):
    """
    dataset: list of sentences
    vocab: corpus, all unique words
    This function takes list of sentences and returns idf values for each word.
    """
    if not isinstance(dataset, list):
        print('Please pass dataset as list of strings')
        return
    idf = dict()
    for word in vocab: # taking each word
        num = 1 + len(dataset) # Total number of sentences
        den = count_sentences_having_word(dataset, word) + 1 # Number of␣
↪senteces having the given word
        result = round(1 + math.log(num/den), 8) # idf value
        idf[word] = result
    return idf

def tf(sentence, word):
    """
    This function takes sentence and word and returns the term frequence of the␣
↪given word
    """
    word_split = Counter(sentence.split(" ")) # Counting each word
    if word_split.get(word):
        return word_split.get(word) / len(word_split) # number of occurance of␣
↪word divided by number of total words
    return 0

def check_values(values):
    """
    This function reduces the size of tfidf values to 50 if it's greater than 50
    """
    if len(values) > 50:
        values = sorted(values, reverse=True)[:50]
```

```
    return values

def check_rows_col(tfidf, rows, col):
    """
    This function matches the size of rows/col to length of tfidf values to␣
 ↪make sparse matrix
    """
    values = numpy.concatenate(tfidf, axis=None)
    if len(values) > len(rows):
        for i in range(len(tfidf) - len(rows)):
            rows.append(0)
            col.append(0)
    else:
        rows = rows[:len(values)]
        col = col[:len(values)]

    return rows, col

def transform(dataset, vocab):
    """
    dataset: list of sentences
    vocab: all unique words
    This function takes list of senteces and all unique words and returns a␣
 ↪sparse matrix having tfidf values for each word of the sentence
    """
    idf_values = get_idf(dataset, vocab) # get the idf value of all the unique␣
 ↪words
    tfidf = list()
    rows, col = list(), list() # These rows and columns are used to make sparse␣
 ↪matrix
    count_row = 0
    for sentence in dataset: # looking into all the sentences one by one.
        values = list()
        count_col = 0
        for word in set(sentence.split(' ')): # taking each word.
            tf_val = tf(sentence, word) # fetching tf of the given word.
            result = tf_val * idf_values.get(word, 0) # calculating the tfidf␣
 ↪value of the given word.
            values.append(result)
            col.append(count_col)
            rows.append(count_row)
            count_col += 1 # increasing the column count for each word
        values =  check_values(values)
        count_row += 1  # increasing the row count for each sentence
        tfidf.append(normalize(numpy.array(values).reshape(1, -1))[0]) #␣
 ↪normalizing and then appending the value to tfidf list.
```

```
        rows, col = check_rows_col(tfidf, rows, col)
    tfidf = numpy.concatenate(tfidf, axis=None) # flattening the list to make␣
↪sparse matrix.
    return csr_matrix((tfidf, (rows, col)), shape=(len(dataset), len(vocab)))
```

[17]:
```
# Write your code here.
# Try not to hardcode any values.
# Make sure its well documented and readble with appropriate comments.
vocab = fit(corpus)
idf = get_idf(corpus, vocab)
tfidf = transform(corpus, vocab)
```

[18]:
```
print(tfidf[745])
```

```
  (0, 0)        0.44808325623776857
  (0, 1)        0.26654021298319613
  (0, 2)        0.44808325623776857
  (0, 3)        0.4344824066870826
  (0, 4)        0.3712755574776605
  (0, 5)        0.44808325623776857
```