Implementing TFIDF Vectorizer

March 30, 2022

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:

$\mathrm{TF}(t) = \frac{\text{Number of times term t appears in a document}}{\text{Total number of terms in the document}}$

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:

$\mathrm{IDF}(t) = \log_e \frac{\text{Total number of documents}}{\text{Number of documents with term t in it}}$ for numerical stabiltiy we will be changing this formula little bit

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

Example

Consider a document containing 100 words wherein the word cat appears 3 times. The term frequency (i.e., TF) for cat is then $\frac{3}{100} = 0.03$. 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 $\log\left(\frac{10000000}{1000}\right) = 4$. Thus, the TF-IDF weight is the product of these quantities: $0.03 \times 4 = 0.12$.

Task-1

1. Build a TF-IDF Vectorizer & compare its results with Sklearn.
   - As a part of this task you will be implementing TF-IDF vectorizer on a collection of text documents.
   - You should compare the results of your own implementation of TF-IDF vectorizer with that of sklearn's implemenation TF-IDF vectorizer.
   - Sklearn does few more tweaks in the implementation of its version of TF-IDF vectorizer, so to replicate the exact results you would need to add following things to your custom implementation of TF-IDF vectorizer:
     1. Sklearn has its vocabulary generated from IDF sorted in alphabetical order
     2. 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. $\text{IDF}(t) = 1 + \log_e \frac{1 + \text{Total number of documents in collection}}{1+\text{Number of documents with term t in it}}$
     3. Sklearn applies L2-normalization on its output matrix.
     4. The final output of sklearn TF-IDF vectorizer is a sparse matrix.
   - Steps to approach this task:
     1. You would have to write both fit and transform methods for your custom implementation of TF-IDF 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 TF-IDF vectorizer.
     3. Print out the IDF values from your implementation and check if its the same as that of sklearn's TF-IDF vectorizer IDF values.
     4. Once you get your vocab and IDF values to be same as that of sklearn's implementation of TF-IDF vectorizer, proceed to the below steps.
     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 here.
     6. After completing the above steps, print the output of your custom implementation and compare it with sklearn's implementation of TF-IDF vectorizer.
     7. To 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.

Note-1: All the necessary outputs of sklearn's TF-IDF 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 sklearn's 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 TF-IDF vectorizer deals with such strings in a different way. To know further details about how sklearn TF-IDF 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]:
```
import warnings
warnings.filterwarnings('ignore')
```

Corpus

[2]:
```
corpus = [
     'this is the first document',
     'this document is the second document',
     'and this is the third one',
     'is this the first document',
]
```

Sklearn Implementation

[3]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

[4]:
```
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus);
```

[5]:
```
skl_output = vectorizer.transform(corpus)
```

The sklearn feature names, they are sorted in alphabetic order by default.

[6]:
```
print(vectorizer.get_feature_names_out())
```

```
['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 method on the corpus the vocab has 9 words in it, and each has its idf value.

[7]:
```
print(vectorizer.idf_)
```

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

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

[8]:
```
print(skl_output.shape)
```

```
(4, 9)
```

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

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

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

The 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.

[10]:
```python
from collections import Counter
from scipy.sparse import csr_matrix
from sklearn.preprocessing import normalize
from tqdm import tqdm

import math
import operator
import numpy as np
```

[11]:
```python
class TFIDFCustom(object):
    """
    This is a custom implementation of TF-IDF Vectorizer.
    """

    def __init__(self, data):
        self.data = data

    def get_unique_words(self):
        """
        This method uniquely identifies the words.
        """
        if isinstance(self.data, list):
            u_words = set()
            for doc in self.data:
                for word in doc.split():
                    if len(word) >= 2:
                        u_words.add(word)
            return sorted(list(u_words))
        else:
            print("Please ensure the data is of list type!")
            return []

    def fit(self):
        """
        This method fits the corpus.
        """
        u_words = self.get_unique_words()
        vocab = {w: i for i, w in enumerate(u_words)}
        return vocab

    def get_tf(self, word, document):
        """
        This method computes term frequency.
        """
        doclist = document.split()
        tf = doclist.count(word) / len(doclist)
        return tf

    def get_idf(self):
        """
        This method computes inverse document frequency.
        """
        idfs = list()
        words = self.get_unique_words()
        N = len(self.data)
        for word in words:
            n = 0
            for doc in self.data:
                if word in doc.split():
                    n += 1
            idf = 1 + np.log((1 + N) / (1 + n))
            idfs.append(idf)
        return np.array(idfs)

    def idf_dict(self):
        """
        This method stores each word and its idf value in a dict.
        """
        words = self.get_unique_words()
        idfs = self.get_idf()
        idf_d = {w: idf for w, idf in zip(words, idfs)}
        return idf_d

    def transform(self, sparse=True):
        """
        This method writes a feature matrix using sparse matrix.
        """
        if isinstance(self.data, list):
            vocab = self.fit()
            idf_d = self.idf_dict()
            sparse_vect = csr_matrix((len(self.data), len(vocab)), dtype=float)
            r = []; c = []; v = [];
            for ri, doc in enumerate(self.data):
                split_doc = doc.split()
                for word in split_doc:
                    if len(word) < 2:
                        continue
                    else:
                        tf = self.get_tf(word=word, document=doc)
                        idf = idf_d.get(word)
                        ci = vocab.get(word, -1)
                        if ci != -1:
                            sparse_vect[ri, ci] = tf * idf
            sparse_vect = normalize(X=sparse_vect)
            if sparse:
                return sparse_vect
            else:
                return sparse_vect.toarray()
        else:
            print("Please ensure the data is of list type!")
```

[12]:
```
tfidf_cus = TFIDFCustom(data=corpus)
```

[13]:
```
display(tfidf_cus.fit())
```

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

[14]:
```
print(tfidf_cus.get_idf())
```

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

[15]:
```
display(tfidf_cus.idf_dict())
```

```
{'and': 1.916290731874155,
 'document': 1.2231435513142097,
 'first': 1.5108256237659907,
 'is': 1.0,
 'one': 1.916290731874155,
 'second': 1.916290731874155,
 'the': 1.0,
 'third': 1.916290731874155,
 'this': 1.0}
```

[16]:
```
print(tfidf_cus.transform(sparse=True))
```

```
  (0, 1)        0.4697913855799205
  (0, 2)        0.580285823684436
  (0, 3)        0.3840852409148149
  (0, 6)        0.3840852409148149
  (0, 8)        0.3840852409148149
  (1, 1)        0.6876235979836937
  (1, 3)        0.2810886740337529
  (1, 5)        0.5386476208856762
  (1, 6)        0.2810886740337529
  (1, 8)        0.2810886740337529
  (2, 0)        0.511848512707169
  (2, 3)        0.267103787642168
  (2, 4)        0.511848512707169
  (2, 6)        0.267103787642168
  (2, 7)        0.511848512707169
  (2, 8)        0.267103787642168
  (3, 1)        0.4697913855799205
  (3, 2)        0.580285823684436
  (3, 3)        0.3840852409148149
  (3, 6)        0.3840852409148149
  (3, 8)        0.3840852409148149
```

Below is `sklearn` sparse matrix.

[17]:
```python
print(skl_output)
```

```
  (0, 8)        0.38408524091481483
  (0, 6)        0.38408524091481483
  (0, 3)        0.38408524091481483
  (0, 2)        0.5802858236844359
  (0, 1)        0.46979138557992045
  (1, 8)        0.281088674033753
  (1, 6)        0.281088674033753
  (1, 5)        0.5386476208856763
  (1, 3)        0.281088674033753
  (1, 1)        0.6876235979836938
  (2, 8)        0.267103787642168
  (2, 7)        0.511848512707169
  (2, 6)        0.267103787642168
  (2, 4)        0.511848512707169
  (2, 3)        0.267103787642168
  (2, 0)        0.511848512707169
  (3, 8)        0.38408524091481483
  (3, 6)        0.38408524091481483
  (3, 3)        0.38408524091481483
  (3, 2)        0.5802858236844359
  (3, 1)        0.46979138557992045
```

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 TF-IDF vectorizer.
- Steps to approach this task:
  1. You would have to write both fit and transform methods for your custom implementation of TF-IDF 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 idf 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 idf values for each term in your vocab.
  3. 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 here.
  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.

[18]:
```
import pickle
```

[19]:
```
with open('cleaned_strings', 'rb') as f:
    corpus = pickle.load(f)

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

```
Number of documents in corpus =  746
```

[20]:
```
class TFIDFMaxFeatures(object):
    """
    This is a custom implementation of TF-IDF Vectorizer with max features.
    """

    def __init__(self, data, max_features):
        self.data = data
        self.max_features = max_features

    def get_unique_words(self):
        """
        This method uniquely identifies the words.
        """
        if isinstance(self.data, list):
            u_words = set()
            for doc in self.data:
                for word in doc.split():
                    if len(word) >= 2:
                        u_words.add(word)
            return sorted(list(u_words))
        else:
            print("Please ensure the data is of list type!")
            return []

    def fit(self):
        """
        This method fits the corpus.
        """
        u_words = self.get_unique_words()
        if len(u_words) > 0:
            idfs = self.get_idf(u_words=u_words)
            vocab = {w: i for w, i in zip(u_words, idfs)}
            vocab = dict(sorted(vocab.items(),
                                key=lambda x:x[1],
                                reverse=True))
            return vocab

    def get_top(self):
        vocab = self.fit()
        top = dict(list(vocab.items())[:self.max_features])
        features = np.array(list(top.keys()))
        return features, list(top.items())

    def get_tf(self, word, document):
        """
        This method computes term frequency.
        """
        doclist = document.split()
        tf = doclist.count(word) / len(doclist)
        return tf

    def get_idf(self, u_words):
        """
        This method computes inverse document frequency.
        """
        idfs = list()
        N = len(self.data)
        for word in u_words:
            n = 0
            for doc in self.data:
                if word in doc.split():
                    n += 1
            idf = 1 + math.log((1 + N) / (1 + n))
            idfs.append(idf)
        return np.array(idfs)

    def transform(self, sparse=True):
        """
        This method writes a feature matrix using sparse matrix.
        """
        if isinstance(self.data, list):
            r = []; c = []; v = [];
            vocab = self.fit()
            features, _ = self.get_top()
            cols = {w: i for i, w in enumerate(features)}
            for ri, doc in enumerate(self.data):
                split_doc = doc.split()
                for word in split_doc:
                    tf = self.get_tf(word=word, document=doc)
                    idf = vocab.get(word)
                    ci = cols.get(word, -1)
                    if ci != -1:
                        r.append(ri)
                        c.append(ci)
                        v.append(tf * idf)
            sparse_vect = csr_matrix(arg1=(v, (r, c)),
                                     shape=(len(self.data), self.max_features))
            sparse_vect = normalize(sparse_vect)
            if sparse:
                return sparse_vect
            else:
                return sparse_vect.toarray()
        else:
            print("Please ensure the data is of list type!")
```

[21]:
```
tfidf_cus_max = TFIDFMaxFeatures(data=corpus, max_features=50)
```

[22]:
```
_, top = tfidf_cus_max.get_top()
display(top)
```

```
[('aailiyah', 6.922918004572872),
 ('abandoned', 6.922918004572872),
 ('abroad', 6.922918004572872),
 ('abstruse', 6.922918004572872),
 ('academy', 6.922918004572872),
 ('accents', 6.922918004572872),
 ('accessible', 6.922918004572872),
 ('acclaimed', 6.922918004572872),
 ('accolades', 6.922918004572872),
 ('accurate', 6.922918004572872),
 ('accurately', 6.922918004572872),
 ('achille', 6.922918004572872),
 ('ackerman', 6.922918004572872),
 ('actions', 6.922918004572872),
 ('adams', 6.922918004572872),
 ('add', 6.922918004572872),
 ('added', 6.922918004572872),
 ('admins', 6.922918004572872),
 ('admiration', 6.922918004572872),
 ('admitted', 6.922918004572872),
 ('adrift', 6.922918004572872),
 ('adventure', 6.922918004572872),
 ('aesthetically', 6.922918004572872),
 ('affected', 6.922918004572872),
 ('affleck', 6.922918004572872),
 ('afternoon', 6.922918004572872),
 ('aged', 6.922918004572872),
 ('ages', 6.922918004572872),
 ('agree', 6.922918004572872),
 ('agreed', 6.922918004572872),
 ('aimless', 6.922918004572872),
 ('aired', 6.922918004572872),
 ('akasha', 6.922918004572872),
 ('akin', 6.922918004572872),
 ('alert', 6.922918004572872),
 ('alike', 6.922918004572872),
 ('allison', 6.922918004572872),
 ('allow', 6.922918004572872),
 ('allowing', 6.922918004572872),
 ('alongside', 6.922918004572872),
 ('amateurish', 6.922918004572872),
 ('amaze', 6.922918004572872),
 ('amazed', 6.922918004572872),
 ('amazingly', 6.922918004572872),
 ('amusing', 6.922918004572872),
 ('amust', 6.922918004572872),
 ('anatomist', 6.922918004572872),
 ('angel', 6.922918004572872),
 ('angela', 6.922918004572872),
 ('angelina', 6.922918004572872)]
```

[23]:
```
print(tfidf_cus_max.transform(sparse=True))
```

```
  (0, 30)	1.0
  (68, 24)	1.0
  (72, 29)	1.0
  (74, 31)	1.0
  (119, 33)	1.0
```

```
(135, 3)      0.37796447300922725
(135, 10)     0.37796447300922725
(135, 18)     0.37796447300922725
(135, 20)     0.37796447300922725
(135, 36)     0.37796447300922725
(135, 40)     0.37796447300922725
(135, 41)     0.37796447300922725
(176, 49)     1.0
(181, 13)     1.0
(192, 21)     1.0
(193, 23)     1.0
(216, 2)      1.0
(222, 47)     1.0
(225, 19)     1.0
(227, 17)     1.0
(241, 44)     1.0
(270, 1)      1.0
(290, 25)     1.0
(333, 26)     1.0
(334, 15)     1.0
(341, 43)     1.0
(344, 42)     1.0
(348, 8)      1.0
(377, 37)     1.0
(409, 5)      1.0
(430, 39)     1.0
(457, 45)     1.0
(461, 4)      1.0
(465, 38)     1.0
(475, 35)     1.0
(493, 6)      1.0
(500, 48)     1.0
(548, 0)      0.7071067811865475
(548, 32)     0.7071067811865475
(608, 14)     1.0
(612, 11)     1.0
(620, 46)     1.0
(632, 7)      1.0
(644, 12)     0.7071067811865475
(644, 27)     0.7071067811865475
(664, 28)     1.0
(667, 22)     1.0
(691, 34)     1.0
(697, 9)      1.0
(722, 16)     1.0
```

End of the file.