Assignment :: TF-IDF Implementation

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.

</font>

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. </p> </font>

Task-1

Libraries Import

In [1]:
```
from functools import reduce
from collections import Counter
from tqdm import tqdm
from scipy.sparse import csr_matrix
from sklearn.preprocessing import normalize

import math
import operator
import numpy as np
```

Corpus

In [2]:
```
## 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',]
```

SkLearn Implementation

In [3]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
skl_output = vectorizer.transform(corpus)
```

Task_1_Description

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:
  1. Sklearn has its vocabulary generated from idf sroted 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. $IDF(t) = 1+\log_{e}\frac{1\text{ }+\text{ Total number of documents in collection}} {1+\text{Number of documents with term t in it}}.$
  3. Sklearn applies L2-normalization on its output matrix.
  4. The final output of sklearn tfidf 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 tfidf vectorizer.
  2. Print out the alphabetically sorted voacb 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 idf values from your implementation and check if its the same as that of sklearns tfidf vectorizer idf values.
  4. Once you get your voacb and idf values to be same as that of sklearns implementation of tfidf 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 https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html
  6. After completing the above steps, print the output of your custom implementation and compare it with sklearns implementation of tfidf 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 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.

T1_Custom_Implementation

In [4]:

```
def fit(text):
    """
    Description: This function is created for performing the below objectives:
        1. Generating the words count against every docuemnt/sentence/row
        2. Creating VOCAB containing all the unique words
        3. Calculating the Inverse Document Frequencies

    Input Parameter: It accepts only one input:
        1. `text`: list
                List of senetences or raw text.

    Output: It returns below parameters:
        1. `vocab`: dict
                Containing all the unique words from the documents
        2. `words_occs_in_doc`: list
                Counter of words for every document
        3. `tot_occs`: dict
                Containing the total number of documents where a specific word is pr
        4. `idf_vals`: dict
                Inverse Document Frequency of every unique word present in the VOCAB
    """
    if isinstance(text,(list,)):
        vocab = set()
        doc_lengths = []
        words_occs_in_doc = []

        # Generating the words count against every docuemnt/sentence/row and creatin
        for i, line in enumerate(tqdm(text)):
            doc_as_list = line.split(" ")

            doc_len = len(doc_as_list)
            doc_lengths.append(doc_len)

            docs_counter = Counter(doc_as_list)
            words_occs_in_doc.append(docs_counter)

            for word in doc_as_list:
                vocab.add(word)

        # Creating a dictionary with the total number of documents where a specific
        vocab = sorted(vocab)
        tot_docs = []
        for word in vocab:
            word_in_docs = []
            for doc in words_occs_in_doc:
                occs_in_docs = doc.get(word,0)
                if occs_in_docs > 1:
                    occs_in_docs = 1
                word_in_docs.append(occs_in_docs)
            word_in_total_docs = reduce(lambda x,y:x+y,word_in_docs)
            tot_docs.append(word_in_total_docs)

        # Calculating the Inverse Document Frequencies
        idfs = []
        for i in range(len(vocab)):
            idf = 1 + np.log((1+len(doc_lengths))/(1+tot_docs[i]))
            idfs.append(np.round(idf,6))
        tot_occs = {word:word_in_docs for word,word_in_docs in zip(vocab,tot_docs)}
        idf_vals = {word:idf for word,idf in zip(vocab,idfs)}
        vocab = {val:i for i,val in enumerate(vocab)}

        return vocab, words_occs_in_doc, tot_occs, idf_vals
    else:
        print("Kindly provide the LIST type input text")
        return None, None, None, None
```

In [5]:

```
vocab, words_occs, total_occs, idf_values  = fit(corpus)
```

```
100%|████████████████████████████████████████████████████████████████████
███| 4/4 [00:00<00:00, 1332.58it/s]
```

Task1_Comparison_with_Sklearn

Task1_Fit_Function

CASE-I

In [6]:

```
# 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']
```

In [7]:

```
# Output from custom implementation
list(vocab.keys())
```

Out[7]:

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

Bingo!! Results matched here

CASE-II

In [8]:

```
# Here we will print the sklearn tfidf vectorizer idf values after applying the fit
# After using the fit function on the corpus the vocab has 9 words in it, and each h

print(vectorizer.idf_)
```

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

In [9]:

```
# # Output from custom implementation
print(list(idf_values.keys()))
print(list(idf_values.values()))
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
[1.916291, 1.223144, 1.510826, 1.0, 1.916291, 1.916291, 1.0, 1.916291, 1.0]
```

Bingo!! Good here as well

In [10]:

```
vocab, words_occs, total_occs, idf_values
```

Out[10]:

```
({'and': 0,
  'document': 1,
  'first': 2,
  'is': 3,
  'one': 4,
  'second': 5,
  'the': 6,
  'third': 7,
  'this': 8},
 [Counter({'this': 1, 'is': 1, 'the': 1, 'first': 1, 'document': 1}),
  Counter({'this': 1, 'document': 2, 'is': 1, 'the': 1, 'second': 1}),
  Counter({'and': 1, 'this': 1, 'is': 1, 'the': 1, 'third': 1, 'one': 1}),
  Counter({'is': 1, 'this': 1, 'the': 1, 'first': 1, 'document': 1})],
 {'and': 1,
  'document': 3,
  'first': 2,
  'is': 4,
  'one': 1,
  'second': 1,
  'the': 4,
  'third': 1,
  'this': 4},
 {'and': 1.916291,
  'document': 1.223144,
  'first': 1.510826,
  'is': 1.0,
  'one': 1.916291,
  'second': 1.916291,
  'the': 1.0,
  'third': 1.916291,
  'this': 1.0})
```

In [11]:

```
def transform(text, vocab, words_occs, total_occs, idf_values):
    """
    Description: This function is created for calculating the Tf-IDF Weights.

    Input Parameter: It accepts 5 inputs:
        1. `text`: list
                List of senetences or raw text.
        2. `vocab`: dict
                Containing all the unique words from the documents
        3. `words_occs_in_doc`: list
                Counter of words for every document
        4. `tot_occs`: dict
                Containing the total number of documents where a specific word is pr
        5. `idf_vals`: dict
                Inverse Document Frequency of every unique word present in the VOCAB

    NOTE:: `vocab`, `words_occs`, `total_occs` and `idf_values` are the outputs from

    Output: It returns the Tf-IDF weights in the form of sparse matrix:
        1. `tfidf_matrix`: csr_matrix
                Tf-IDF weights
    """
    if isinstance(text,(list,)):
        tfidf_vals = []
        row = []
        col = []
        # Traversing every document to calulate the term frequency in a single docum
        for i,val in enumerate(tqdm(text)):
            doc_as_list = text[i].split(" ")
            tfidf_val = []
            doc_as_list = set(doc_as_list)
            # Using the calculation results of fit function in this loop to compute
            for word in list(doc_as_list):
                row.append(i)
                col.append(vocab.get(word,0.0))
                tot_words_in_doc = len(doc_as_list)
                word_occs_in_doc = words_occs[i].get(word,0)
                tf = word_occs_in_doc/len(doc_as_list)
                idf = idf_values.get(word,0.0)
                tfidf = tf * idf
                tfidf_val.append(tfidf)
            tfidf_l2_normed = normalize(np.array(tfidf_val).reshape(-1,1),axis=0)
            tfidf_vals.append(tfidf_l2_normed)

        # Generating the sparse matrix of TF-IDF weights
```

```
            values = [v1 for val in tfidf_vals for v1 in val]
            values = np.array(values).flatten()
            row, col = np.array(row).flatten(), np.array(col).flatten()
            tfidf_matrix = csr_matrix((values,(row,col)),shape=(len(text),len(list(vocab

            return tfidf_matrix
        else:
            print("Kindly provide the LIST type input text")
            return None
```

In [12]:

```
tfidf_matrix_values = transform(corpus, vocab, words_occs, total_occs, idf_values)
```

```
100%|████████████████████████████████████████████████████████████████████████████████
███| 4/4 [00:00<00:00, 1333.43it/s]
```

Task1_Transform_Function

CASE-I

In [13]:

```
# Sklearn output
print(skl_output[0])
```

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

In [14]:

```
# Custom implementation output
tfidf_values = tfidf_matrix_values.toarray()
print(tfidf_values[0][tfidf_values[0] != 0])
```

```
[0.46979148 0.58028587 0.38408518 0.38408518 0.38408518]
```

CASE-II

In [15]:

```
# Sklearn output
print(skl_output[1])
```

```
  (0, 8)	0.281088674033753
  (0, 6)	0.281088674033753
  (0, 5)	0.5386476208856763
  (0, 3)	0.281088674033753
  (0, 1)	0.6876235979836938
```

In [16]:

```
# Custom implementation output
print(tfidf_values[1][tfidf_values[1] != 0])
```

```
[0.6876237  0.28108861 0.53864758 0.28108861 0.28108861]
```

CASE-III

In [17]:

```
# Sklearn output
print(skl_output[2])
```

```
  (0, 8)	0.267103787642168
  (0, 7)	0.511848512707169
  (0, 6)	0.267103787642168
  (0, 4)	0.511848512707169
  (0, 3)	0.267103787642168
  (0, 0)	0.511848512707169
```

In [18]:

```
# Custom implementation output
print(tfidf_values[2][tfidf_values[2] != 0])
```

```
[0.51184853 0.26710376 0.51184853 0.26710376 0.51184853 0.26710376]
```

CASE-IV

In [19]:
```
# Sklearn output
print(skl_output[3])
```

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

In [20]:
```
# Custom implementation output
print(tfidf_values[3][tfidf_values[3] != 0])
```

```
[0.46979148 0.58028587 0.38408518 0.38408518 0.38408518]
```

So, good here as well!!

Creating the DataFrame of the Tf-Idf Weights for every document

In [21]:
```
import pandas as pd
```

In [22]:
```
tfidf_weights_df = pd.DataFrame(tfidf_values,
                                columns=list(total_occs.keys()),
                                index=['Doc/Sentence/Row-1','Doc/Sentence/Row-2','Do

tfidf_weights_df
```

Out[22]:

	and	document	first	is	one	second	the	third
Doc/Sentence/Row-1	0.000000	0.469791	0.580286	0.384085	0.000000	0.000000	0.384085	0.000000
Doc/Sentence/Row-2	0.000000	0.687624	0.000000	0.281089	0.000000	0.538648	0.281089	0.000000
Doc/Sentence/Row-3	0.511849	0.000000	0.000000	0.267104	0.511849	0.000000	0.267104	0.511849
Doc/Sentence/Row-4	0.000000	0.469791	0.580286	0.384085	0.000000	0.000000	0.384085	0.000000

Task-2

Task_2_Description

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:

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 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.
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 https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html
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.

In [23]:

```
# 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
```

T2_Custom_Implementation

In [24]:

```
def task2_fit(text,top_idf_scr=50):
    """
    Description: This function is created for performing the below objectives:
        1. Generating the words count against every docuemnt/sentence/row
        2. Creating VOCAB containing all the unique words
        3. Calculating the Inverse Document Frequencies

    Input Parameter: It accepts below inputs:
        1. `text`: list
                List of senetences or raw text.
        2. `top_idf_scr`: int
                Number of top words to be selected based on IDF values.

    Output: It returns below parameters:
        1. `vocab`: dict
                Containing all the unique words from the documents
        2. `words_occs_in_doc`: list
                Counter of words for every document
        3. `tot_occs`: dict
                Containing the total number of documents where a specific word is pr
        4. `idf_vals`: dict
                Inverse Document Frequency of every unique word present in the VOCAB
    """
    if isinstance(text,(list,)):
        vocab = []
        doc_lengths = []
        words_occs_in_doc = []

        # Generating the words count against every docuemnt/sentence/row and creatin
        for i, line in enumerate(tqdm(text)):
            doc_as_list = line.split(" ")

            doc_len = len(doc_as_list)
            doc_lengths.append(doc_len)

            docs_counter = Counter(doc_as_list)
            words_occs_in_doc.append(docs_counter)

            result = [vocab.append(word) for word in doc_as_list if word not in voca

        # Creating a dictionary with the total number of documents where a specific
        tot_docs = []
        for word in vocab:
            word_in_docs = []
            for doc in words_occs_in_doc:
                occs_in_docs = doc.get(word,0)
                if occs_in_docs > 1:
                    occs_in_docs = 1
                word_in_docs.append(occs_in_docs)
            word_in_total_docs = reduce(lambda x,y:x+y,word_in_docs)
            tot_docs.append(word_in_total_docs)

        # Calculating the Inverse Document Frequencies
        idfs = []
        for i in range(len(vocab)):
            idf = 1 + np.log((1+len(doc_lengths))/(1+tot_docs[i]))
            idfs.append(np.round(idf,6))

        tot_occs = {word:word_in_docs for word,word_in_docs in zip(vocab,tot_docs)}
        idf_vals = {word:idf for word,idf in zip(vocab,idfs)}
        top_idf_scores = dict(sorted(idf_vals.items(),key=operator.itemgetter(1),rev
        vocab = {val:i for i,val in enumerate(list(top_idf_scores.keys()))}
        top_tot_occs = {k: v for k, v in tot_occs.items() if k in list(vocab.keys())

        return vocab, words_occs_in_doc, top_tot_occs, top_idf_scores
    else:
        print("Kindly provide the LIST type input text")
        return None, None, None, None
```

Observations

- When we use set() or convert the list into a set then it changes the order of the elements due to this I used a list to store the words with the order preserved. **Hence, I implemented the order in which the documents are processed, thus the top 50 idf values are selected based on the order of processed rows as well not only by random sorting. This way I was able to give preference to processed order as well along with the Idf value.**
  - Referred link: https://stackoverflow.com/questions/9792664/converting-a-list-to-a-set-changes-element-order
- While working on this assignment, I came to know about an useful property of built-in ***operator*** :: ***operator*** is a built-in module providing a set of convenient operators. In two words ***operator.itemgetter(n)*** constructs a callable that assumes an iterable object (e.g. list, tuple, set) as input, and fetches the nth element out of it.
  - So, using this I was able to sort the dictionary based on the IDF values and able to perform the slicing as well to select the top 50 words with highest value.
    - Referred link: https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-value

- Referred link: https://stackoverflow.com/questions/18595686/how-do-operator-itemgetter-and-sort-work

```
In [25]: vocab, words_occs, total_occs, idf_values = task2_fit(corpus)
```

```
100%|███████████████████████████████████████| 746/746 [00:00<00:00, 1897.16it/s]
```

Task_2_Results

```
In [26]: len(list(vocab.values()))

Out[26]: 50
```

```
In [27]: len(list(idf_values.keys()))

Out[27]: 50
```

```
In [28]: len(list(total_occs.keys()))

Out[28]: 50
```

```
In [39]: vocab
```

```
Out[39]: {'aimless': 0,
          'distressed': 1,
          'drifting': 2,
          'nearly': 3,
          'attempting': 4,
          'artiness': 5,
          'existent': 6,
          'gerardo': 7,
          'emptiness': 8,
          'effort': 9,
          'messages': 10,
          'buffet': 11,
          'science': 12,
          'teacher': 13,
          'baby': 14,
          'owls': 15,
          'florida': 16,
          'muppets': 17,
          'person': 18,
          'overdue': 19,
          'screenplay': 20,
          'post': 21,
          'practically': 22,
          'structure': 23,
          'tightly': 24,
          'constructed': 25,
          'vitally': 26,
          'occurs': 27,
          'content': 28,
          'fill': 29,
          'dozen': 30,
          'highest': 31,
          'superlative': 32,
          'require': 33,
          'puzzle': 34,
          'solving': 35,
          'fit': 36,
          'pulls': 37,
          'punches': 38,
          'graphics': 39,
          'number': 40,
          'th': 41,
          'insane': 42,
          'massive': 43,
          'unlockable': 44,
          'properly': 45,
          'aye': 46,
          'rocks': 47,
          'doomed': 48,
          'conception': 49}
```

In [29]:

```
tfidf_matrix_values = transform(corpus, vocab, words_occs, total_occs, idf_values)
```

```
100%|████████████████████████████████████████████████████████████████████████| 
746/746 [00:00<00:00, 1871.96it/s]
```

In [30]:

```
tfidf_values = tfidf_matrix_values.toarray()
```

In [36]:

```
tfidf_values[0].shape, tfidf_values[0]
```

Out[36]:

```
((50,),
 array([0.57735027, 0.57735027, 0.57735027, 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ]))
```

Task_2_Results_as_DF

Creating the DataFrame of the Tf-Idf Weights for every document

In [31]:

```
import pandas as pd
```

In [32]:

```
print(list(total_occs.keys()))
```

```
['aimless', 'distressed', 'drifting', 'nearly', 'attempting', 'artiness', 'existen
t', 'gerardo', 'emptiness', 'effort', 'messages', 'buffet', 'science', 'teacher', 'b
aby', 'owls', 'florida', 'muppets', 'person', 'overdue', 'screenplay', 'post', 'prac
tically', 'structure', 'tightly', 'constructed', 'vitally', 'occurs', 'content', 'fi
ll', 'dozen', 'highest', 'superlative', 'require', 'puzzle', 'solving', 'fit', 'pull
s', 'punches', 'graphics', 'number', 'th', 'insane', 'massive', 'unlockable', 'prope
rly', 'aye', 'rocks', 'doomed', 'conception']
```

In [33]:

```
tfidf_weights_df = pd.DataFrame(tfidf_values,
                                columns=list(total_occs.keys()))

tfidf_weights_df
```

Out[33]:

	aimless	distressed	drifting	nearly	attempting	artiness	existent	gerardo	emptiness	effort
0	0.57735	0.57735	0.57735	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
1	0.00000	0.00000	0.00000	1.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
2	0.00000	0.00000	0.00000	0.0	0.57735	0.57735	0.57735	0.0	0.0	0.0
3	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
4	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...
741	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0

	aimless	distressed	drifting	nearly	attempting	artiness	existent	gerardo	emptiness	effort
742	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
743	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
744	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0
745	0.00000	0.00000	0.00000	0.0	0.00000	0.00000	0.00000	0.0	0.0	0.0

746 rows × 50 columns