Q5

October 25, 2019

Problem 5

Text Mining with Cosine Similarities

To begin, the document must be processed (a). In order to do so, the following six steps need be performed. [1] Convert to lower case, [2] remove stop words, [3] remove punctuation, [4] remove singular characters, [5] stem all words, and [6] replace digits with their english representation.

```
In [5]: import re
        import os
        import nltk
        from nltk.stem import PorterStemmer
        from nltk.stem import LancasterStemmer
        import pandas as pd
        from num2words import num2words
        import ast
        import csv
        import numpy as np
        import queue
        import time
        import json
```

Building a Tool Set

Processing documents

In order to process many documents in a clean and readable fashion, each document will be treated as an object. This document class is shown below and incorporates a pre-process functionality with the rules mentioned above. Furthermore, there is a similar pre-process function to be called on pure text as opposed to a file.

```
In [6]: class Document:

            def __init__(self, filename = '', path = '', text = ''):
                self.filename = filename
                self.path = path
                self.text = text

            def process_query(self):


                stop_words = ['ourselves', 'hers', 'between', 'yourself', 'but', 'again', 'the
```

```
        porter = PorterStemmer()

        query = self.text

        # Tokenize
        query = re.sub(r'\W+', ' ', query).split()
        # apply lower case
        query = [x.lower() for x in query]
        # remove stop words
        query = [i for i in query if not i in stop_words]
        # remove single characters
        query = [i for i in query if len(i) > 1]
        # stemming
        tmp = []
        for word in query:
            tmp.append(porter.stem(word))
        query = tmp

        # convert numbers
        index = 0
        for word in query:
            if word.isdigit():
                query[index] = num2words(word)
            index += 1

        return query


    def pre_process(self):
        """
        Pre-processor function to remove and tokenize indivual document
        object for prepartion of analysis. Returns processed data.
        """

        stop_words = ['ourselves', 'hers', 'between', 'yourself', 'but', 'again', 'the
        porter = PorterStemmer()

        with open(os.path.join(path, self.filename), encoding="utf8", errors="surrogat

            data = text.read()

            # Tokenize
            data = re.sub(r'\W+', ' ', data).split()
            # apply lower case
            data = [x.lower() for x in data]
            # remove stop words
            data = [i for i in data if not i in stop_words]
            # remove single characters
```

```
        data = [i for i in data if len(i) > 1]
        # stemming
        tmp = []
        for word in data:
            tmp.append(porter.stem(word))
        data = tmp

        # convert numbers
        index = 0
        for word in data:
            if word.isdigit():
                data[index] = num2words(word)
            index += 1

        print("Document ready!")

        return data
```

Calculating TF-IDF Vectors

Simlar to the document class, the TFIDFVector class will generate a vector object from data parameters such as document frequencies and universal word totals. This object maintains the ability to convert itself into an TF-IDF vector coressponding to an individual document. Similar to the case in the document class a seperate vectorize function is used for text based queries.

```
In [7]: class TFIDFVector:
            def __init__(self, document, N, DF):
                """
                Creates document vector to compute TF-IDF.
                For document vectorization, document should
                be a dictionary with word counts. For a query
                document should be in a string format.
                """
                self.document = document
                self.N = N
                self.DF = DF


            def vectorize(self):
                """
                Function generates TF-IDF Vector for
                individual document in corpus.
                """
                tf_idf_vector = {}
                for term in self.document:
                    # term frequency
                    count = self.document[term]
                    tf = count/len(self.document)
                    # document frequency
```

```
            df = sum(self.DF[term])
            # inverse document frequency
            idf = np.log(self.N/(df + 1))
            # TF-IDF
            tf_idf = tf*idf
            save = {term : tf_idf}
            tf_idf_vector.update(save)

        return tf_idf_vector

    def query_vectorize(self):

        tf_idf_vector_query = {}

        for term in self.document:
            count = counter(self.document, term)[1]
            tf = count/len(self.document)
            # document frequency
            df = sum(self.DF[term])
            # inverse document frequency
            idf = np.log(self.N/(df + 1))
            # TF-IDF
            tf_idf = tf*idf
            save = {term : tf_idf}
            tf_idf_vector_query.update(save)

        return tf_idf_vector_query
```

Utility Functions

Each function below is used to help calculate individual data parameters. the precise purpose of each function is described in its documentation.

```
In [15]: def main(directory):
             """
             Main driver function, takes in directory and returns term counts for
             each individual document.
             """
             docs = []
             for entry in entries:
                 docs.append(Document(entry, path))

             processed = []
             for document in docs:
                  processed.append(document.pre_process())

             processed_counts = termCounts(processed)
```

```
    with open('wordCounts.txt', 'w') as file:
        file.write(json.dumps(processed_counts))

    return processed_counts


def counter(lst, term):
    """
    Utility function to count redundant occurences
    in a list.
    """
    count = 0
    for ele in lst:
        if (ele == term):
            count = count + 1
    return term, count


def termCounts(corpus):
    """
    Takes list of pre-processed documents (corpus) and returns individual
    term counts for each document.
    """
    count = 0
    termCount_corpus = []
    for document in corpus:
        termCounts_doc = {}
        tmp = []
        print("Counting... " + str(count))
        for term in document:
            tmp.append(counter(document,term))
        terms = set(tmp)
        terms = list(terms)
        termCounts_doc.update(terms)
        termCount_corpus.append(termCounts_doc)
        print(len(termCount_corpus))
        count += 1
    return termCount_corpus




def universe_size(data):
    """
    Utility function to compute and return
    term universe size 'N'.
    """
    N = 0
```

```
    for doc in data:
        n=0
        for term in doc:
            count = doc[term]
            n += count
        N += n
    return N

def document_frequency(data):
    """
    Utility function to compute document frequency for
    all terms in pre-processed corpus.
    """
    DF = {}
    for i in range(len(data)):
        tokens = data[i]
        for w in tokens:
            try:
                DF[w].add(i)
            except:
                DF[w] = {i}
    return DF
```

Cosine Similarity

Cosine similarity allows for a metric to compare to n-dimensional vectors landing between the values of 0 and 1. This similarity is quantified by the tightness of the angle between any two vectors. The function below performs this calculation for every possible pair of documents in the data set (250 x 250)!

```
In [15]: def cosineSim(data):
             """
             Function to compute the pairwise cosine similarity
             throughout all documents. Produces an n x n matrix
             saved as a csv. n is number of documents.

             """

             start_time = time.time()

             # Initialize all tf-idf vectors
             start_vectors  = []
             print('Initializing...')
             print()

             # Append each tfd-idf vector to start_vectors (all docs)
             for i in range(len(data)):
                 start_vectors.append(TFIDFVector(data[i], universe_size(data), document_freque
```

```python
# Queue up vectors for comparison
to_compare = queue.Queue(maxsize = len(data))
cols = ["col" + str(i) for i in range(len(data))]

for vector in start_vectors:
    to_compare.put(vector)

print("Vectors ready")
print()
print("Working...")
tick = 0

# Start comparisons
sims = pd.DataFrame()

# Outer loop handles queued vectors
for vector in start_vectors:
    base = to_compare.get()
    print()
    column = []

    # Inner loop compares each vector in data to current vector in the queue
    for i in range(len(data)):

        # Convert pair of vectors to data frame
        comparison = pd.DataFrame([base, start_vectors[i]])

        # Replace non-present words from opposing document with 0 values
        comparison.fillna(0, inplace = True)
        a = comparison.iloc[0]
        b = comparison.iloc[1]

        # Convert to numpy vectors
        a = a.to_numpy()
        b = b.to_numpy()

        # manually compute cosine similarity
        dot = np.dot(a,b)
        norma = np.linalg.norm(a)
        normb = np.linalg.norm(b)
        cos = round(dot / (norma * normb), 5)
        column.append(cos)

    # Insert pair wise comparison as column in final matrix
    insertion = pd.Series(column)
    sims.insert(tick, cols[tick], insertion)
```

```
        tick += 1
        print("#####################################")
        print("Computation complete for vector " + str(tick) + "/" + str(len(data)))
        print("#####################################")



    export_csv = sims.to_csv (r'/home/ian/Dropbox/School/Current_courses/Data_Mining/a


    print("--- %s minutes ---" % round(((time.time() - start_time)/60), 2))
```

```
In [15]: def retrieval(queries, data):
             """
             Function performs a comparison
             analysis between TF-IDF vectorized queries
             and all documents in the data parameter. Data
             must be in pre-processed format.
             """

             start_time = time.time()

             # Initialize all tf-idf vectors
             print()
             print('Initializing data...')

             start_vectors  = []

             # Append each tfd-idf vector to start_vectors (all docs)
             for i in range(len(data)):
                 start_vectors.append(TFIDFVector(data[i], universe_size(data), document_freque

             # Queue up vectors for comparison
             to_compare = queue.Queue(maxsize = len(data))
             cols = ["col" + str(i) for i in range(len(data))]

             for vector in start_vectors:
                 to_compare.put(vector)

             print('Intializing queries...')

             # Pre-process queries
             processed_queries = []
             for query in queries:
```

```
    filename = ''
    q = Document(filename, path, text = query)
    processed_queries.append(q.process_query())


# Vectorize queries
query_vectors = []
for i in range(len(queries)):
    query_vectors.append(TFIDFVector(processed_queries[i], universe_size(data), do

print('Queries processed!')

# Comparisons


# Queue up vectors for comparison
to_search = queue.Queue(maxsize = len(query_vectors))
cols = ["col" + str(i) for i in range(len(data))]

for vector in query_vectors:
    to_search.put(vector)

retrievals = pd.DataFrame()

tick = 0
# Outer loop handles queued vectors

print("Searching...")
for vector in query_vectors:
    base = to_search.get()
    column_retrievals = []

    # Inner loop compares each vector in data to current vector in the queue
    for i in range(len(data)):

        # Convert pair of vectors to data frame
        comparison = pd.DataFrame([base, start_vectors[i]])

        # Replace non-present words from opposing document with 0 values
        comparison.fillna(0, inplace = True)
        a = comparison.iloc[0]
        b = comparison.iloc[1]

        # Convert to numpy vectors
        a = a.to_numpy()
        b = b.to_numpy()

        # manually compute cosine similarity
```

```
                dot = np.dot(a,b)
                norma = np.linalg.norm(a)
                normb = np.linalg.norm(b)
                cos = round(dot / (norma * normb), 5)
                column_retrievals.append(cos)

            # Insert pair wise comparison as column in final matrix
            insertion = pd.Series(column_retrievals)
            retrievals.insert(tick, cols[tick], insertion)
            tick += 1
        print('Done!')
        print("--- %s seconds ---" % round(((time.time() - start_time)), 2))

        return retrievals
```

```
In [6]: ####################
        #### PRE-PROCESS ####
        ####################
        """
        This section of code combines the above functions
        and classes to generate the necessary term counts
        and document frequencies to compute individual TF-IDF
        vectors for idividual documents. Run to process new
        data. Data is saved to file 'termCounts.txt'.
        """

        # Directory path
        path = '/home/ian/Dropbox/School/Current_courses/Data_Mining/assignment_3/Data'

        # Data directory
        entries = os.listdir(path)

        # Uncomment if file not already created
        results = main(entries)
```

```
In [38]: ###############
         ### ANALYSIS ###
         ###############

         with open('wordCounts.txt', 'r') as file:
             data = file.read()
             data = ast.literal_eval(data)


         data;
```

In [18]:
```python
df = pd.read_csv('matrix.csv')
df
# Run cosineSim() if not been run already (run time ~ 75 minutes)
# matrix = cosineSim(data)
```

Out[18]:
```
        col0     col1     col2     col3     col4     col5     col6     col7  \
0    1.00000  0.13074  0.15071  0.05337  0.07692  0.05715  0.03246  0.13397
1    0.13074  1.00000  0.42906  0.10437  0.13335  0.15244  0.06316  0.33175
2    0.15071  0.42906  1.00000  0.14807  0.17398  0.15496  0.07850  0.45667
3    0.05337  0.10437  0.14807  1.00000  0.06599  0.06573  0.03951  0.15739
4    0.07692  0.13335  0.17398  0.06599  1.00000  0.07362  0.03496  0.19353
5    0.05715  0.15244  0.15496  0.06573  0.07362  1.00000  0.04047  0.17547
6    0.03246  0.06316  0.07850  0.03951  0.03496  0.04047  1.00000  0.09304
7    0.13397  0.33175  0.45667  0.15739  0.19353  0.17547  0.09304  1.00000
8    0.07524  0.12327  0.16531  0.08372  0.10675  0.06948  0.04752  0.19088
9    0.09731  0.16180  0.18229  0.08062  0.08951  0.08303  0.04191  0.21597
10   0.14021  0.20744  0.25638  0.07014  0.13943  0.10950  0.05952  0.30852
11   0.02357  0.06311  0.09649  0.03624  0.05851  0.03884  0.02965  0.12875
12   0.08719  0.12794  0.17085  0.06999  0.08295  0.07648  0.05533  0.17643
13   0.06038  0.08462  0.10103  0.03159  0.04157  0.03752  0.02317  0.12649
14   0.14240  0.12440  0.15397  0.21649  0.06960  0.06421  0.04997  0.16593
15   0.07975  0.10839  0.12881  0.05522  0.05348  0.05503  0.04411  0.12265
16   0.04708  0.12851  0.18337  0.11757  0.14834  0.06788  0.04322  0.23519
17   0.06407  0.08127  0.10457  0.05257  0.03777  0.04130  0.02312  0.11070
18   0.04047  0.05792  0.09894  0.05002  0.02423  0.04579  0.03008  0.10223
19   0.14748  0.08439  0.11982  0.05164  0.05155  0.05395  0.02894  0.18960
20   0.05025  0.11157  0.15699  0.06707  0.15787  0.06506  0.04344  0.17419
21   0.02349  0.04908  0.06191  0.01580  0.08027  0.02935  0.09264  0.06092
22   0.08782  0.12176  0.14018  0.06705  0.06383  0.06318  0.05049  0.14998
23   0.04387  0.08481  0.11905  0.05345  0.09947  0.04983  0.02615  0.11690
24   0.03710  0.06286  0.08373  0.02839  0.05035  0.04314  0.01651  0.07749
25   0.06916  0.10820  0.11774  0.06198  0.04488  0.06044  0.03990  0.13566
26   0.11839  0.17773  0.23393  0.11749  0.08429  0.09063  0.04820  0.24368
27   0.03520  0.10019  0.14108  0.07594  0.10897  0.11820  0.03700  0.16958
28   0.06159  0.08573  0.11083  0.04877  0.04021  0.05102  0.03282  0.12743
29   0.07896  0.11944  0.13519  0.06674  0.11139  0.06386  0.04203  0.18301
..       ...      ...      ...      ...      ...      ...      ...      ...
219  0.05518  0.11884  0.15940  0.07528  0.08327  0.07264  0.04782  0.17133
220  0.14481  0.22488  0.24101  0.06842  0.11344  0.09812  0.05753  0.23947
221  0.11180  0.18244  0.19717  0.06674  0.13085  0.08512  0.05442  0.24166
222  0.02117  0.02323  0.02458  0.01038  0.02015  0.01217  0.00824  0.02379
223  0.13403  0.19872  0.22568  0.09583  0.11528  0.10652  0.08350  0.24187
224  0.10826  0.15350  0.18711  0.14203  0.09225  0.09372  0.05126  0.25063
225  0.16192  0.23974  0.27827  0.09297  0.14599  0.12124  0.08330  0.32918
226  0.13025  0.18054  0.20011  0.05927  0.12900  0.09161  0.05770  0.21120
227  0.07852  0.11597  0.14529  0.07407  0.06043  0.06531  0.04073  0.15349
228  0.10988  0.15328  0.16118  0.07635  0.10532  0.08171  0.07528  0.19315
229  0.07851  0.12420  0.15513  0.08956  0.06294  0.06058  0.03183  0.19187
```

230	0.08083	0.10078	0.12953	0.07269	0.05765	0.05969	0.04769	0.14527
231	0.03807	0.06431	0.08257	0.02891	0.03934	0.02893	0.02030	0.08811
232	0.06112	0.09882	0.13616	0.05422	0.05126	0.04781	0.04522	0.15251
233	0.07830	0.15326	0.16967	0.08732	0.09575	0.09181	0.05777	0.19515
234	0.15240	0.10249	0.12326	0.16402	0.07053	0.06025	0.03410	0.14300
235	0.04696	0.08492	0.09530	0.04203	0.04993	0.04304	0.02949	0.11234
236	0.08914	0.11275	0.14697	0.08291	0.05262	0.06599	0.04856	0.15098
237	0.11050	0.17155	0.19186	0.08560	0.09131	0.09097	0.06045	0.22880
238	0.20560	0.45880	0.54053	0.16144	0.17808	0.17410	0.10234	0.50739
239	0.15486	0.50410	0.54739	0.13060	0.16040	0.15040	0.08173	0.43341
240	0.12591	0.20103	0.23365	0.09218	0.10332	0.12113	0.08749	0.25874
241	0.09221	0.13672	0.16000	0.05921	0.08538	0.07376	0.04751	0.17991
242	0.11318	0.14929	0.18468	0.07375	0.05690	0.07731	0.04556	0.17737
243	0.02616	0.06043	0.07802	0.03756	0.03659	0.04388	0.03665	0.11308
244	0.13777	0.13403	0.14582	0.04925	0.05757	0.06658	0.05219	0.15914
245	0.00555	0.24070	0.21752	0.01871	0.01401	0.02480	0.01117	0.03475
246	0.10723	0.21832	0.25073	0.09496	0.11778	0.10961	0.05418	0.26929
247	0.05787	0.11449	0.13587	0.06481	0.11069	0.06009	0.04220	0.18833
248	0.10710	0.18694	0.23150	0.08932	0.13752	0.09927	0.05506	0.28217

	col8	col9	...	col239	col240	col241	col242	col243	\
0	0.07524	0.09731	...	0.15486	0.12591	0.09221	0.11318	0.02616	
1	0.12327	0.16180	...	0.50410	0.20103	0.13672	0.14929	0.06043	
2	0.16531	0.18229	...	0.54739	0.23365	0.16000	0.18468	0.07802	
3	0.08372	0.08062	...	0.13060	0.09218	0.05921	0.07375	0.03756	
4	0.10675	0.08951	...	0.16040	0.10332	0.08538	0.05690	0.03659	
5	0.06948	0.08303	...	0.15040	0.12113	0.07376	0.07731	0.04388	
6	0.04752	0.04191	...	0.08173	0.08749	0.04751	0.04556	0.03665	
7	0.19088	0.21597	...	0.43341	0.25874	0.17991	0.17737	0.11308	
8	1.00000	0.09629	...	0.16176	0.17275	0.08042	0.12434	0.04587	
9	0.09629	1.00000	...	0.20069	0.15127	0.10650	0.09937	0.05922	
10	0.15532	0.16094	...	0.28290	0.21424	0.17613	0.17506	0.07749	
11	0.04465	0.03885	...	0.07777	0.05428	0.04503	0.04058	0.02607	
12	0.09895	0.09492	...	0.18317	0.14080	0.11831	0.12243	0.05117	
13	0.04616	0.07013	...	0.10221	0.10123	0.05960	0.05852	0.02822	
14	0.09032	0.09586	...	0.13621	0.15873	0.09953	0.10405	0.05052	
15	0.14184	0.06999	...	0.14242	0.14690	0.10031	0.09416	0.03490	
16	0.09451	0.07082	...	0.16309	0.11809	0.08425	0.09268	0.05658	
17	0.05541	0.06557	...	0.09491	0.09836	0.05729	0.07279	0.02444	
18	0.08565	0.03406	...	0.07702	0.09178	0.05243	0.06516	0.07202	
19	0.05350	0.07653	...	0.11007	0.10118	0.05653	0.06234	0.04137	
20	0.11288	0.07848	...	0.13620	0.13544	0.09079	0.07560	0.06904	
21	0.03737	0.02690	...	0.05454	0.04737	0.02472	0.02649	0.01469	
22	0.15020	0.10244	...	0.15216	0.17775	0.09068	0.12052	0.04625	
23	0.08045	0.12665	...	0.09532	0.09423	0.06254	0.06729	0.02434	
24	0.02551	0.08360	...	0.08106	0.06040	0.13140	0.03421	0.01785	
25	0.08439	0.08582	...	0.12060	0.16002	0.06591	0.06848	0.03730	
26	0.09941	0.12990	...	0.21293	0.19073	0.11857	0.12718	0.05237	

27	0.09112	0.06588	...	0.12082	0.11247	0.05753	0.06079	0.03454
28	0.05706	0.06403	...	0.10533	0.08951	0.05941	0.06459	0.03640
29	0.11546	0.08218	...	0.15512	0.12991	0.09390	0.08614	0.05033
..	...	...	...	...	...	...	...	...
219	0.09306	0.10255	...	0.13814	0.14258	0.09502	0.07760	0.04070
220	0.15823	0.15032	...	0.28219	0.21168	0.14147	0.16885	0.05392
221	0.14578	0.13269	...	0.22245	0.19249	0.15931	0.13047	0.06180
222	0.01889	0.02125	...	0.02528	0.03038	0.02081	0.02964	0.00700
223	0.18893	0.16117	...	0.25545	0.27414	0.16558	0.19128	0.08656
224	0.13281	0.13164	...	0.19200	0.17815	0.10874	0.11231	0.05181
225	0.17307	0.18311	...	0.29971	0.27793	0.18513	0.23150	0.09019
226	0.16069	0.11918	...	0.20791	0.19211	0.12973	0.15506	0.05398
227	0.07447	0.10606	...	0.14362	0.12047	0.08029	0.08547	0.03580
228	0.19108	0.10980	...	0.18657	0.18596	0.10144	0.13209	0.05645
229	0.06742	0.11676	...	0.14577	0.10871	0.08410	0.08115	0.03214
230	0.09647	0.06950	...	0.12506	0.13033	0.09059	0.11189	0.04753
231	0.05057	0.04103	...	0.07101	0.06410	0.04974	0.04176	0.02125
232	0.09836	0.07964	...	0.12648	0.10922	0.06906	0.07457	0.03278
233	0.09749	0.09399	...	0.18914	0.14797	0.10435	0.09937	0.04589
234	0.07587	0.07398	...	0.11790	0.13217	0.07085	0.07948	0.03431
235	0.05312	0.06173	...	0.10350	0.11485	0.06533	0.05749	0.02937
236	0.06667	0.08964	...	0.14631	0.13191	0.06535	0.15872	0.03513
237	0.20614	0.12162	...	0.20585	0.19802	0.12125	0.17317	0.06054
238	0.23398	0.23775	...	0.56994	0.33215	0.20075	0.22919	0.10435
239	0.16176	0.20069	...	1.00000	0.24572	0.17472	0.17626	0.08405
240	0.17275	0.15127	...	0.24572	1.00000	0.14386	0.17700	0.08838
241	0.08042	0.10650	...	0.17472	0.14386	1.00000	0.10250	0.06147
242	0.12434	0.09937	...	0.17626	0.17700	0.10250	1.00000	0.04285
243	0.04587	0.05922	...	0.08405	0.08838	0.06147	0.04285	1.00000
244	0.09731	0.09822	...	0.15709	0.14591	0.09158	0.13446	0.03721
245	0.01108	0.00917	...	0.28689	0.01630	0.01413	0.01706	0.00791
246	0.16015	0.16614	...	0.25589	0.19279	0.12986	0.12680	0.06370
247	0.11140	0.10291	...	0.17852	0.11850	0.08729	0.05085	0.03812
248	0.13064	0.16906	...	0.24116	0.25155	0.13901	0.13416	0.09227

	col244	col245	col246	col247	col248
0	0.13777	0.00555	0.10723	0.05787	0.10710
1	0.13403	0.24070	0.21832	0.11449	0.18694
2	0.14582	0.21752	0.25073	0.13587	0.23150
3	0.04925	0.01871	0.09496	0.06481	0.08932
4	0.05757	0.01401	0.11778	0.11069	0.13752
5	0.06658	0.02480	0.10961	0.06009	0.09927
6	0.05219	0.01117	0.05418	0.04220	0.05506
7	0.15914	0.03475	0.26929	0.18833	0.28217
8	0.09731	0.01108	0.16015	0.11140	0.13064
9	0.09822	0.00917	0.16614	0.10291	0.16906
10	0.17048	0.02464	0.20380	0.16687	0.30571
11	0.03566	0.01103	0.05951	0.03865	0.06632

12	0.11661	0.01310	0.14236	0.06443	0.12798
13	0.06257	0.00560	0.07459	0.04067	0.08662
14	0.12976	0.01238	0.11279	0.13280	0.13602
15	0.12943	0.01176	0.11170	0.04829	0.08238
16	0.06454	0.02804	0.14832	0.11145	0.17155
17	0.10297	0.00797	0.08359	0.05183	0.10260
18	0.04487	0.00759	0.05801	0.03329	0.07629
19	0.06118	0.00945	0.07636	0.14846	0.10509
20	0.04954	0.01573	0.10588	0.07955	0.10870
21	0.02582	0.00817	0.03975	0.04786	0.04923
22	0.10609	0.01138	0.12400	0.07461	0.12658
23	0.06480	0.00657	0.11701	0.05994	0.08753
24	0.03669	0.01079	0.05917	0.02506	0.08631
25	0.08546	0.01116	0.09755	0.05025	0.08830
26	0.13445	0.01985	0.14425	0.09007	0.18721
27	0.05019	0.01737	0.09133	0.08139	0.12810
28	0.09016	0.01462	0.07143	0.04304	0.08637
29	0.08821	0.03069	0.10184	0.11401	0.14244
..	...	...	...	...	...
219	0.05440	0.01458	0.10061	0.06972	0.10765
220	0.15262	0.03349	0.19640	0.14578	0.21316
221	0.11815	0.02204	0.18978	0.13892	0.21159
222	0.01962	0.00303	0.03120	0.02048	0.02842
223	0.15128	0.02802	0.21457	0.12155	0.22696
224	0.11237	0.02279	0.13658	0.07757	0.15388
225	0.19534	0.02696	0.23442	0.16604	0.28723
226	0.12704	0.02205	0.15146	0.12653	0.19308
227	0.08473	0.01559	0.10037	0.05678	0.10372
228	0.18672	0.01848	0.14119	0.10768	0.21201
229	0.07894	0.01360	0.10066	0.06976	0.12683
230	0.08806	0.01254	0.11357	0.06291	0.10005
231	0.04258	0.00987	0.05678	0.04028	0.05339
232	0.08121	0.01441	0.11264	0.05617	0.09196
233	0.07384	0.01333	0.10933	0.08813	0.12550
234	0.08484	0.00970	0.09149	0.07136	0.09610
235	0.05246	0.04233	0.07891	0.04708	0.08391
236	0.09489	0.00976	0.09158	0.04385	0.08751
237	0.13727	0.01420	0.17526	0.10859	0.15802
238	0.20589	0.12633	0.29845	0.16134	0.28408
239	0.15709	0.28689	0.25589	0.17852	0.24116
240	0.14591	0.01630	0.19279	0.11850	0.25155
241	0.09158	0.01413	0.12986	0.08729	0.13901
242	0.13446	0.01706	0.12680	0.05085	0.13416
243	0.03721	0.00791	0.06370	0.03812	0.09227
244	1.00000	0.01842	0.11455	0.07383	0.12822
245	0.01842	1.00000	0.02737	0.00712	0.01867
246	0.11455	0.02737	1.00000	0.11443	0.19421
247	0.07383	0.00712	0.11443	1.00000	0.15795

```
248  0.12822  0.01867  0.19421  0.15795  1.00000

[249 rows x 249 columns]
```

Query Processing

In order to search the data set and find a best matching document to some query, the query itself must be 'vectorized' (i.e. compute a TF-IDF vector) and that vector must be compared via the cosine similarity to all other documents in the data set. Once completed the results for each query are sorted by cosine similarity output from largest to smallest. We the take the top 10 results and display them below.

```
In [19]: queries = ["""Once upon a time . . .
         there were three little pigs,
         who left their mummy and daddy to see the
         world.""",
         """There once lived a poor tailor, who had a son called Aladdin,
         a careless, idle boy who would
         do nothing but play all day long
         in the streets with little idle boys like himself."""]
```

```
In [39]: df = retrieval(queries, data)
```

```
Initializing data...
Intializing queries...
Queries processed!
Searching...
Done!
--- 45.13 seconds ---
```

```
In [53]: df["Document"] = entries

         df['query1'] = df["col0"]
         df['query2'] = df["col1"]
         df.drop(['col0', 'col1'], axis=1)

         query1 = pd.DataFrame(df['Document'])
         query1['Score'] = df['query1']

         query2 = pd.DataFrame(df['Document'])
         query2['Score'] = df['query2']

         five_d1 = query1.sort_values(by=['Score'], ascending=False).head(n=10)
         five_d2 = query2.sort_values(by=['Score'], ascending=False).head(n=10)
```

Results

The resultant scores for each query given are displayed in the data frames below. As can be seen, the query returned the most relavent documents by highest score. The validity of this result can be checked by observing the titles of the highest scoring documents!

In [54]:
```
# Query 1
five_d1
```

Out[54]:

	Document	Score
43	3lpigs.txt	0.40721
238	enginer.txt	0.14670
7	gulliver.txt	0.14432
27	lgoldbrd.txt	0.14432
160	empty.txt	0.13223
81	ccm.txt	0.13131
2	5orange.txt	0.12428
178	hound-b.txt	0.11972
197	darkness.txt	0.11508
70	abbey.txt	0.11485

In [55]:
```
# Query 2
five_d2
```

Out[55]:

	Document	Score
97	alad10.txt	0.35603
28	adv_alad.txt	0.33792
116	tinsoldr.txt	0.16191
147	yukon.txt	0.12163
202	fantasy.txt	0.12065
197	darkness.txt	0.10851
160	empty.txt	0.10630
236	snowmaid.txt	0.10380
146	lmtchgrl.txt	0.10281
53	aesop11.txt	0.09160