Simple Shakespeare Search Engine Assignment

PART I: Data Exploration

Question 1: Read file "shakespeare_small.json" directly from the url to a dataframe : http://elmokhtari.com/downloads/ds8003/shakespeare_small.json

```
import json
import requests
```

All we were doing here is reading the small .json file from the URL and printing it to a pyspark dataframe.

```
r = requests.get("http://elmokhtari.com/downloads/ds8003/shakespeare_small.json")
df = sqlContext.createDataFrame([json.loads(line) for line in r.iter_lines()])
df.show()
```

Output:

```
>>> df.show()
+---+-------+-----------+---------+-------------+-------------+--------------------+----+
|_id|line_id|line_number|play_name|      speaker|speech_number|          text_entry|type|
+---+-------+-----------+---------+-------------+-------------+--------------------+----+
|  3|      4|      1.1.1| Henry IV|KING HENRY IV|            1|So shaken as we a...|line|
|  4|      5|      1.1.2| Henry IV|KING HENRY IV|            1|Find we a time fo...|line|
|  5|      6|      1.1.3| Henry IV|KING HENRY IV|            1|And breathe short...|line|
|  6|      7|      1.1.4| Henry IV|KING HENRY IV|            1|To be commenced i...|line|
|  7|      8|      1.1.5| Henry IV|KING HENRY IV|            1|No more the thirs...|line|
|  8|      9|      1.1.6| Henry IV|KING HENRY IV|            1|Shall daub her li...|line|
|  9|     10|      1.1.7| Henry IV|KING HENRY IV|            1|Nor more shall tr...|line|
| 10|     11|      1.1.8| Henry IV|KING HENRY IV|            1|Nor bruise her fl...|line|
| 11|     12|      1.1.9| Henry IV|KING HENRY IV|            1|Of hostile paces:...|line|
| 12|     13|     1.1.10| Henry IV|KING HENRY IV|            1|Which, like the m...|line|
| 13|     14|     1.1.11| Henry IV|KING HENRY IV|            1|All of one nature...|line|
| 14|     15|     1.1.12| Henry IV|KING HENRY IV|            1|Did lately meet i...|line|
| 15|     16|     1.1.13| Henry IV|KING HENRY IV|            1|And furious close...|line|
| 16|     17|     1.1.14| Henry IV|KING HENRY IV|            1|Shall now, in mut...|line|
| 17|     18|     1.1.15| Henry IV|KING HENRY IV|            1|March all one way...|line|
| 18|     19|     1.1.16| Henry IV|KING HENRY IV|            1|Against acquainta...|line|
| 19|     20|     1.1.17| Henry IV|KING HENRY IV|            1|The edge of war, ...|line|
| 20|     21|     1.1.18| Henry IV|KING HENRY IV|            1|No more shall cut...|line|
| 21|     22|     1.1.19| Henry IV|KING HENRY IV|            1|As far as to the ...|line|
| 22|     23|     1.1.20| Henry IV|KING HENRY IV|            1|Whose soldier now...|line|
+---+-------+-----------+---------+-------------+-------------+--------------------+----+
only showing top 20 rows
```

Question 2: Show the rows count.

There is a built-in count function for dataframes.

```
df.count()
```

```
>>> df.count()
68
```

Question 3: Upload the file: shakespeare_full.json to your linux machine and load its content to a dataframe df2.

We begin by moving the file from the VM into HDFS

```
hadoop fs -mkdir m1thanabalasingam
hadoop fs -put shakespeare_full.json m1thanabalasingam
```

We import the .json file into a separate dataframe, df2.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df2 =
sqlContext.read.json("/user/maria_dev/m1thanabalasingam/shakespeare_full.json")
df2.show()
```

```
>>> df2.show()
+---+-------+-----------+---------+-------------+-------------+--------------------+-----+
|_id|line_id|line_number|play_name|      speaker|speech_number|          text_entry| type|
+---+-------+-----------+---------+-------------+-------------+--------------------+-----+
|  0|      1|           | Henry IV|             |         null|               ACT I|  act|
|  1|      2|           | Henry IV|             |         null|SCENE I. London. ...|scene|
|  2|      3|           | Henry IV|             |         null|Enter KING HENRY,...| line|
|  3|      4|      1.1.1| Henry IV|KING HENRY IV|            1|So shaken as we a...| line|
|  4|      5|      1.1.2| Henry IV|KING HENRY IV|            1|Find we a time fo...| line|
|  5|      6|      1.1.3| Henry IV|KING HENRY IV|            1|And breathe short...| line|
|  6|      7|      1.1.4| Henry IV|KING HENRY IV|            1|To be commenced i...| line|
|  7|      8|      1.1.5| Henry IV|KING HENRY IV|            1|No more the thirs...| line|
|  8|      9|      1.1.6| Henry IV|KING HENRY IV|            1|Shall daub her li...| line|
|  9|     10|      1.1.7| Henry IV|KING HENRY IV|            1|Nor more shall tr...| line|
| 10|     11|      1.1.8| Henry IV|KING HENRY IV|            1|Nor bruise her fl...| line|
| 11|     12|      1.1.9| Henry IV|KING HENRY IV|            1|Of hostile paces:...| line|
| 12|     13|     1.1.10| Henry IV|KING HENRY IV|            1|Which, like the m...| line|
| 13|     14|     1.1.11| Henry IV|KING HENRY IV|            1|All of one nature...| line|
| 14|     15|     1.1.12| Henry IV|KING HENRY IV|            1|Did lately meet i...| line|
| 15|     16|     1.1.13| Henry IV|KING HENRY IV|            1|And furious close...| line|
| 16|     17|     1.1.14| Henry IV|KING HENRY IV|            1|Shall now, in mut...| line|
| 17|     18|     1.1.15| Henry IV|KING HENRY IV|            1|March all one way...| line|
| 18|     19|     1.1.16| Henry IV|KING HENRY IV|            1|Against acquainta...| line|
| 19|     20|     1.1.17| Henry IV|KING HENRY IV|            1|The edge of war, ...| line|
+---+-------+-----------+---------+-------------+-------------+--------------------+-----+
only showing top 20 rows
```

Question 4: Show the count of entries grouped by “speaker” on df2.

We use the groupBy function to aggregate the data into showing the number of times a speaker appeared in *shakespeare_full.json*

```
df2.groupBy('speaker').count().show()
```

```
>>> df2.groupBy("speaker").count().show()
+-----------------+-----+
|          speaker|count|
+-----------------+-----+
|       EUPHRONIUS|   16|
|Third Conspirator|   12|
|            PETER|   63|
|  First Gentleman|  284|
|           AEGEON|  150|
|        DONALBAIN|   10|
|        LYCHORIDA|   11|
|          QUINTUS|   30|
|           AENEAS|  153|
|           Porter|   97|
|          RUTLAND|   26|
|              NYM|   78|
|   LORD FITZWATER|   27|
|         CARDINAL|  120|
|       Attendants|    2|
|       ANTIPHOLUS|    6|
|    Third Servant|   31|
|        ANNE PAGE|   31|
|        Moonshine|    6|
|       SIR ANDREW|  155|
+-----------------+-----+
only showing top 20 rows
```

Question 5: Using spark.sql, show all entries where line_number starts with “1.1.” and text_entry contains the word “sometimes”.

We specifically only need the rows where the words ‘sometimes’ appears, as well as line_numbers from section 1.1.

```
question5select = df2.select(df2['_id'], df2['speaker'],
df2['line_number'], df2['text_entry'])

question5filter = question5.where(df2.text_entry.contains('sometimes')
& df2.line_number.startswith('1.1'))
```

```
>>> question5filter.show()
+-----+---------+-----------+--------------------+
|  _id|  speaker|line_number|          text_entry|
+-----+---------+-----------+--------------------+
|18634|    PHILO|     1.1.63|Sir, sometimes, w...|
|32496|  HORATIO|     1.1.59|Did sometimes mar...|
|61534| BASSANIO|    1.1.166|Of wondrous virtu...|
|64418|  SLENDER|    1.1.240|A justice of peac...|
|75845|ANTIOCHUS|     1.1.34|Yon sometimes fam...|
+-----+---------+-----------+--------------------+
```

Question 6: Generate a list with the number of characters in every text entry where the speaker is "DONALBAIN"

We first take the rows where the speaker is Donalbain, and then use the length function to determine the number of characters in each entry. The output is a list, so we use the flatMap function to print out the column as a list.

```
donalbain = df2.where(df2.speaker == "DONALBAIN")
lengths = donalbain.withColumn('Length', F.length('text_entry'))
lengthList = lengths.select("Length").rdd.flatMap(lambda x:
x).collect()
lengthList
```

```
>>> lengthList
[14, 47, 15, 45, 11, 28, 36, 43, 49, 18]
```

Question 7: Consider all text entries of the speaker "DONALBAIN". Generate a list of pairs (key, value) where **key** is the _id of the text entry and **value** is the number of words in this text entry.

Using the lengths dataframe from Question6, we introduce a new column that concatenates the *_id* and calculated *Length* together. Since this result also needs to be a list, we use flatMap again to convert our column into a list.

```
textDetails = lengths.withColumn('keyValue', F.concat(F.lit('('),
                                 F.col('_id'), F.lit(','),
                                 F.col('Length'), F.lit(')')))
keyValueList = textDetails.select("keyValue").rdd.flatMap(lambda x:
x).collect()
keyValueList
```

```
>>> keyValueList
[u'(56668,14)', u'(56698,47)', u'(56699,15)', u'(56700,45)', u'(56701,11)', u'(
56702,28)', u'(56723,36)', u'(56724,43)', u'(56725,49)', u'(56726,18)']
```

PART II

Question 2.1: [Building Index] Compute TFIDF scores for all words in all text entries and build an inverted index. This index will be stored in the dataframe **tokensWithTfIdf** containing the following columns: (token, _id, tf, df, idf, tf_idf).

- **token** is any word in text entries
- **_id**: text entry id,
- **(tf,idf,tf_idf)** scores of the pair (token,_id).

This function was given by the professor and is being used to clean the text_entry column. The function removes all punctuations and makes all the letters lowercase.

```
def lower_clean_str(x):
       punc = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
       lowercased_str = x.lower()
       for ch in punc:
              lowercased_str = lowercased_str.replace(ch, '')
       return lowercased_str
```

These libraries were all necessary to help in calculating the tf-idf value of each (_id, token) pair.

```
from pyspark.sql import SQLContext
from pyspark.sql.functions import UserDefinedFunction, split, explode,
col, log10
from pyspark.sql.types import StringType
from pyspark.ml.feature import Tokenizer
```

We begin by importing the shakespeare_full.json file from the URL.

```
sqlContext = SQLContext(sc)
data =
sqlContext.read.json("/user/maria_dev/m1thanabalasingam/shakespeare_fu
ll.json")
```

We only need the _id and text_entry columns, so we will place those into another dataframe, and clean the text_entry column using our predefined function above.

```
requiredData = data.select('_id', 'text_entry')
udf = UserDefinedFunction(lambda x: lower_clean_str(x), StringType())
requiredData = requiredData.withColumn('CleanText',
udf(requiredData.text_entry))
```

We need to tokenize each line so we can set up our final tf-idf dataframe. Tokenizer allows us to transform each text_entry into a list of the words

```
tokenizer = Tokenizer(inputCol='CleanText', outputCol='words_token')
requiredData_tokenized =
tokenizer.transform(requiredData).select('_id', 'words_token')
```

We can explode the new column, words_token, and use it alongside _id to create tokensWithTfIdf. We can use this to calculate all of the metrics required for the TF-IDF.

```
tokensWithTfIdf =
requiredData_tokenized.select(explode('words_token'), '_id')
```

We're going to rename the col column to token, so it's easier to comprehend.

```
tokensWithTfIdf = tokensWithTfIdf.withColumnRenamed('col', 'token')
```

We start by finding tf. We need to count how many times a term appears in a text_entry. This is done by grouping each (_id, token) pair, and returning their count. If a token appears more than once in a text_entry, this count will show a value greater than one. In most cases, the token probably appears once, so this column will have a lot of ones in it.

NOTE: The slides had two different definitions for calculating TF. We used the definition: the number of times a term appears in a document, from Lecture 11, Slide 18. We are not dividing by the number of terms within a document, as is shown in Lecture11, Slide 17.

```
tfNum = tokensWithTfIdf.groupBy('_id', 'token').count()
tokensWithTfIdf = tokensWithTfIdf.join(tfNum, on = ['_id', 'token'])
tokensWithTfIdf = tokensWithTfIdf.withColumnRenamed('count', 'TF')
```

We drop the columns we have no use for anymore, since we don't need these in our final dataframe.

```
tokensWithTfIdf = tokensWithTfIdf.drop('TFWordCount')
tokensWithTfIdf = tokensWithTfIdf.drop('TFIDWordCount')
```

IDF is calculated by finding the log(base 10) of the total number of text_entries, divided by the number of documents a token appears in. This is done by finding the distinct (_id, token) pairs, and grouping by token would return a count of the number of documents (_id) that had that token at least once. We can say at least once because we only selected the distinct pairs, so our groupBy statement only occurs when docFrequency has only distinct (_id, token) pairs.

```
docFrequency = tokensWithTfIdf.select('_id', 'token').distinct()
docFrequency = docFrequency.groupBy('token').count()
```

We join the df column to the tokensWithTfIdf dataframe.

```
tokensWithTfIdf = tokensWithTfIdf.join(docFrequency, on = ['token'])
tokensWithTfIdf= tokensWithTfIdf.withColumnRenamed('count', 'df')
```

To calculate the number of total documents, we just need to get a count of all the _ids, since each one stood for a different text. We then can do our IDF calculation.

```
totalDocs = requiredData.count()
tokensWithTfIdf = tokensWithTfIdf.withColumn('IDF', log10(totalDocs /
col('df')))
```

Now that we have TF and IDF, all we have to do is multiply those two columns together, and we get out TF-IDF.

```
tokensWithTfIdf = tokensWithTfIdf.withColumn('TF-IDF',
tokensWithTfIdf['TF'] * tokensWithTfIdf['IDF'])
```

As our final dataframe is, we have duplicate (_id, token) pairs. This is redundant because we've already accounted for terms appearing more than once with the TF-IDF calculation. So let's get rid of our duplicates and then we will have our completed dataframe.

```
tokensWithTfIdf = temp.dropDuplicates(['_id', 'token'])
tokensWithTfIdf.show()
```

```
>>> tokensWithTfIdf.show()
+-------------+------+---+---+------------------+------------------+
|        token|   _id| TF| df|               IDF|            TF-IDF|
+-------------+------+---+---+------------------+------------------+
|    abbeygate| 24185|  1|  1| 5.046869596500938| 5.046869596500938|
| accumulation| 20133|  1|  1| 5.046869596500938| 5.046869596500938|
|      acheron| 57256|  1|  3| 4.569748341781275| 4.569748341781275|
|      acheron| 98368|  1|  3| 4.569748341781275| 4.569748341781275|
|      acheron| 68418|  1|  3| 4.569748341781275| 4.569748341781275|
|apprehensions| 54089|  1|  3| 4.569748341781275| 4.569748341781275|
|apprehensions| 73798|  1|  3| 4.569748341781275| 4.569748341781275|
|apprehensions|100531|  1|  3| 4.569748341781275| 4.569748341781275|
|    arguments| 10407|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 50054|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 54859|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 41460|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments|104534|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 10940|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 43529|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments|104086|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments| 81497|  1| 10| 4.046869596500938| 4.046869596500938|
|    arguments|104418|  1| 10| 4.046869596500938| 4.046869596500938|
|          art| 23109|  1|829|2.1283150659506638|2.1283150659506638|
|          art| 43877|  1|829|2.1283150659506638|2.1283150659506638|
+-------------+------+---+---+------------------+------------------+
only showing top 20 rows
```

Question 2.2: [Search] Given a query and a value N, retrieve the top N matching text entries with their score (use TFIDF scores to retrieve the matching text entries). Construct a function **search_words (query, N)** where query is a string and N an integer. The result will display the top N text entries ordered by their score in descending order. Show the results of each of the following queries, show three sets of results N=1, 3, 5:

```
query1 = "to be or not"
query2 = "so far so"
query = "if you said so"
```

We are not importing anything new in this question because we're assuming we are continuing from 2.1. We are using those same libraries to complete this section.

```
def search_words(query, N):
```

Transform the query into a list and calculate the number of words in the query, which we will use later.

```
    queryWords = query.split()
    numWords = len(queryWords)
```

Let's find the instances where the queryWords are in tokensWithTfIdf

```
    temp =
tokensWithTfIdf.filter(tokensWithTfIdf.token.isin(queryWords))
```

Initialize our final dataframe with the _id and its associated text_entry.

```
    finaldf = requiredData.select('_id', 'text_entry')
```

numQueryWords will represent the part of the score equation that shows the number of DISTINCT times the query words appear in a text_entry. temp only consists of instances that the query words appear in any text_entry, so it's simply a matter of grouping by _id. We then join that aggregated column to finaldf for further processing.

```
    numQueryWords = temp.groupBy('_id').count()
    finaldf = finaldf.join(numQueryWords, on = ['_id'])
```

tfidfSum is the summation of TF-IDFs for each id. Again, since all we have in temp is distinct instances of the query words appearing in each text_entry, and their associated TF-IDF values, all we need to do is group by id and sum the TF-IDF values. We can then join this column to finaldf for our final calculation.

```
    tfidfSum = temp.groupBy('_id').sum('TF-IDF')
    finaldf = finaldf.join(tfidfSum, on = ['_id'])
```

We do each step of our calculation, beginning with the fraction portion: the part that will divide the number of query words that appeared in the text_entry, divided by the total number of words in the query, numWords.

```
finaldf = finaldf.withColumn('part1', col('count')/numWords)
```

We calculate the score by multiplying the two parts of the equation, part 1 and the summation. We round the score to three decimal places, as requested in the question.

```
finaldf = finaldf.withColumn('score', round((finaldf['part1'] *
finaldf['sum(TF-IDF)']), 3))
```

We can drop the three columns that were added to assist in calculations, because we don't need these in our final answer.

```
finaldf = finaldf.drop('count')
finaldf = finaldf.drop('sum(TF-IDF)')
finaldf = finaldf.drop('part1')
```

We rearrange the columsn to be in the same order as mentioned in the question.

```
finaldf = finaldf.select('_id', 'score', 'text_entry')
```

Since we want our top N values, we're going to sort by score in descending order, and return the top N results.

```
finaldf = finaldf.sort('score', ascending = False)
finaldf.show(N)
return finaldf.show(N)
```

```
>>> search_words("to be or not", 1)
```

```
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|34229|6.946|To be, or not to ...|
+-----+-----+--------------------+
only showing top 1 row
```

```
>>> search_words("to be or not", 3)
```

```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 34229|6.946|To be, or not to ...|
|103117|6.135|will not be seen;...|
|109930|6.045|Not like a corse;...|
+------+-----+--------------------+
only showing top 3 rows
```

```
>>> search_words("to be or not", 5)
```

```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 34229|6.946|To be, or not to ...|
|103117|6.135|will not be seen;...|
|109930|6.045|Not like a corse;...|
| 64679|4.899|to meddle or make...|
|101007|4.899|Or else you love ...|
+------+-----+--------------------+
only showing top 5 rows
```

```
>>> search_words("so far so", 1)
```

```
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|68413|3.593|And so far am I g...|
+-----+-----+--------------------+
only showing top 1 row
```

```
>>> search_words("so far so", 3)
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|68413|3.593|And so far am I g...|
|11154|3.593|So do I wish the ...|
|51283|2.764|so, so, so. Well ...|
+-----+-----+--------------------+
only showing top 3 rows
```

```
>>> search_words("so far so", 5)
```

```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 68413|3.593|And so far am I g...|
| 11154|3.593|So do I wish the ...|
| 51283|2.764|so, so, so. Well ...|
| 96897|2.671|That brought her ...|
|110732|2.671|nothing, let him ...|
+------+-----+--------------------+
only showing top 5 rows
```

```
>>> search_words("if you said so",1)
+-----+------+--------------------+
|  _id| score|          text_entry|
+-----+------+--------------------+
|18430|11.773|of an If, as, If ...|
+-----+------+--------------------+
only showing top 1 row
```

```
>>> search_words("if you said so",3)
+-----+------+--------------------+
|  _id| score|          text_entry|
+-----+------+--------------------+
|18430|11.773|of an If, as, If ...|
|29571|  6.37|If you but said s...|
|61123| 5.089|O, did you so? An...|
+-----+------+--------------------+
only showing top 3 rows
```

```
>>> search_words("if you said so", 5)
+------+------+--------------------+
|   _id| score|          text_entry|
+------+------+--------------------+
| 18430|11.773|of an If, as, If ...|
| 29571|  6.37|If you but said s...|
| 61123| 5.089|O, did you so? An...|
|106075| 5.073|And if it please ...|
| 10471| 4.364|You said so much ...|
+------+------+--------------------+
only showing top 5 rows
```

Question 2.3: [Job] Write a file **search.py** that you will run using **spark-submit**. This file will contain parts of the code from questions 1 and 2 and additional code as you deem necessary. The file search.py will generate the same results as question 2.

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import UserDefinedFunction, split, explode,
col, log10, round
from pyspark.sql.types import StringType
from pyspark.ml.feature import Tokenizer
```

NOTE: The three functions used in this file that are not *main* were the functions used in part 1 and 2. The code and associated comments are, for the most part, repeating what is above. Changes have been highlighted and explained.

```
def lower_clean_str(x):
	punc = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
	lowercased_str = x.lower()
	for ch in punc:
		lowercased_str = lowercased_str.replace(ch, '')
	return lowercased_str
```

In *search_words*, this time around I am including *query* and *N* as inputs to the function. For question 2, I manually tested each combination for the sake of screenshots, but for this question I felt that automating this process would make more sense.

```
def search_words(query, N, tokensWithTfIdf, requiredData):
```

Transform the query into a list and calculate the number of words in the query, which we will use later.

```
	queryWords = query.split()
	numWords = len(queryWords)
```

Let's find the instances where the queryWords are in tokensWithTfIdf

```
	temp =
tokensWithTfIdf.filter(tokensWithTfIdf.token.isin(queryWords))
```

Initialize our final dataframe with the _id and its associated text_entry

```
	finaldf = requiredData.select('_id', 'text_entry')
```

numQueryWords will represent the part of the score equation that shows the number of DISTINCT times the query words appear in a text_entry. temp1 only consists of instances that the query words appear in any text_entry, so it's simply a matter of grouping by _id. We then join that aggregated column to finaldf for further processing

```
        numQueryWords = temp.groupBy('_id').count()
        finaldf = finaldf.join(numQueryWords, on = ['_id'])
```

tfidfSum is the summation of TF-IDFs for each id. Again, since all we have in temp1 is distinct instances of the query words appearing in each text_entry, and their associated TF-IDF values, all we need to do is group by id and sum the TF-IDF values. We can then join this column to finaldf for our final calculation.

```
        tfidfSum = temp.groupBy('_id').sum('TF-IDF')
        finaldf = finaldf.join(tfidfSum, on = ['_id'])
```

We do each step of our calculation, beginning with the fraction portion: the part that will divide the number of query words that appeared in the text_entry, divided by the total number of words in the query, numWords

```
        finaldf = finaldf.withColumn('part1', col('count')/numWords)
```

We calculate the score by multiplying the two parts of the equation, part 1 and the summation. We round the score to three decimal places, as requested in the question.

```
        finaldf = finaldf.withColumn('score', round((finaldf['part1'] *
finaldf['sum(TF-IDF)']), 3))
```

We can drop the three columns that were added to assist in calculations, because we don't need these in our final answer.

```
        finaldf = finaldf.drop('count')
        finaldf = finaldf.drop('sum(TF-IDF)')
        finaldf = finaldf.drop('part1')
```

We rearrange the columns to be in the same order as mentioned in the question

```
        finaldf = finaldf.select('_id', 'score', 'text_entry')
```

Since we want our top N values, I'm going to sort by score in descending order, and return the top N results.

```
        finaldf = finaldf.sort('score', ascending = False)
        return finaldf.show(N)
```

For Question 2.1, I did not create the TF-IDF within a function; we created it in a pyspark shell. Since we want to execute our program through a *spark-submit* command, I chose to throw all of part 2.1 into a function to create the TFIDF dataframe for me.

`def create_TFIDF(data):`

We only need the _id and text_entry columns, so we will place those into another dataframe, and clean the text_entry column using our predefined function above.

```
        requiredData = data.select('_id', 'text_entry')
        udf = UserDefinedFunction(lambda x: lower_clean_str(x),
StringType())
```

```
        requiredData = requiredData.withColumn('CleanText',
udf(requiredData.text_entry))
```

We need to tokenize each line so we can set up our final tf-idf dataframe. Tokenizer allows us to transform each text_entry into a list of the words

```
        tokenizer = Tokenizer(inputCol='CleanText',
outputCol='words_token')
        requiredData_tokenized =
tokenizer.transform(requiredData).select('_id', 'words_token')
```

We can explode the new column, words_token, and use it alongside _id to create tokensWithTfIdf. We can use this to calculate all of the metrics required for the TF-IDF.

```
        temp = requiredData_tokenized.select(explode('words_token'),
'_id')
```

We're going to rename the col column to token, so it's easier to comprehend.

```
        temp = temp.withColumnRenamed('col', 'token')
```

We start by finding tf. We need to count how many times a term appears in a text_entry. This is done by grouping each (_id, token) pair, and returning their count. If a token appears more than once in a text_entry, this count will show a value greater than one. In most cases, the token probably appears once, so this column will have a lot of ones in it.

```
        tfNum = temp.groupBy('_id', 'token').count()
        temp = temp.join(tfNum, on = ['_id', 'token'])
        temp = temp.withColumnRenamed('count', 'TF')
```

We drop the columns we have no use for anymore, since we don't need these in our final dataframe.

```
        temp = temp.drop('TFWordCount')
        temp = temp.drop('TFIDWordCount')
```

IDF is calculated by finding the log(base 10) of the total number of text_entries, divided by the number of documents a token appears in. This is done by finding the distinct (_id, token) pairs, and grouping by token would return a count of the number of documents (_id) that had that token at least once. We can say at least once because we only selected the distinct pairs, so our groupBy statement only occurs when docFrequency has only distinct (_id, token) pairs.

```
        docFrequency = temp.select('_id', 'token').distinct()
        docFrequency = docFrequency.groupBy('token').count()
```

We join the df column to the tokensWithTfIdf dataframe.

```
        temp = temp.join(docFrequency, on = ['token'])
        temp = temp.withColumnRenamed('count', 'df')
```

To calculate the number of total documents, we just need to get a count of all of the _ids, since each one stood for a different text. We then can do our IDF calculation.

```
        totalDocs = requiredData.count()
        temp = temp.withColumn('IDF', log10(totalDocs / col('df')))
```

Now that we have TF and IDF, all we have to do is multiply those two columns together, and we get our TF-IDF.

```
        temp = temp.withColumn('TF-IDF', temp['TF'] * temp['IDF'])
```

As our final dataframe is, we have duplicate (_id, token) pairs. This is redundant because we've already accounted for terms appearing more than once with the TF-IDF calculation. So let's get rid of our duplicates and then we will have our final TF-IDF dataframe.

```
        tokensWithTfIdf = temp.dropDuplicates(['_id', 'token'])
        return tokensWithTfIdf, requiredData
```

This is the most significant addition for Question 2.3 that was not part of 2.1 or 2.2. The *main* function and the statement afterwards are what allow the code file to operate upon running the *spark-submit* command.

def main(sc):

We begin by importing the shakespeare_full.json file from the URL.

```
        sqlContext = SQLContext(sc)
        data =
sqlContext.read.json("/user/maria_dev/m1thanabalasingam/shakespeare_fu
ll.json")
```

We create the TF-IDF dataframe, and then create the 9 (query, N) combinations we want our program to execute. It then runs the *search_words* function for each pairing and prints the results, as shown below.

```
        tokensWithTfIdf, requiredData = create_TFIDF(data)
        queries = ["to be or not", "so far so", "if you said so"]
        N = [1, 3, 5]
        for i in queries:
                for j in N:
                        print("Current Query:", i)
                        print("Number of Results:", j)
                        search_words(i, j, tokensWithTfIdf, requiredData)

if __name__  == "__main__":
    conf = SparkConf().setAppName("MyApp")
    sc = SparkContext(conf = conf)
    main(sc)
    sc.stop()
```

The code was almost identical to that of parts 1 and 2. The only difference was the main function, where we included the nested for loop to return all of the (query, N) combinations that we had to calculate for 2.2.

```
spark-submit search.py
```

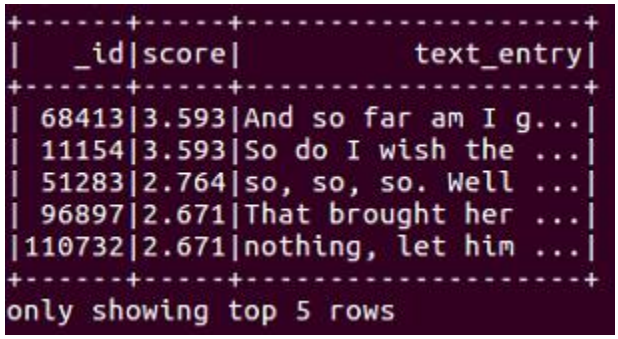
```
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|34229|6.946|To be, or not to ...|
+-----+-----+--------------------+
only showing top 1 row
```

```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 34229|6.946|To be, or not to ...|
|103117|6.135|will not be seen;...|
|109930|6.045|Not like a corse;...|
+------+-----+--------------------+
only showing top 3 rows
```

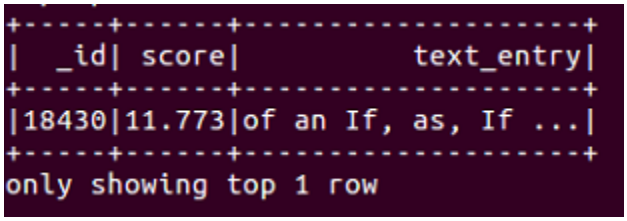
```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 34229|6.946|To be, or not to ...|
|103117|6.135|will not be seen;...|
|109930|6.045|Not like a corse;...|
| 64679|4.899|to meddle or make...|
|101007|4.899|Or else you love ...|
+------+-----+--------------------+
only showing top 5 rows
```

```
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|68413|3.593|And so far am I g...|
+-----+-----+--------------------+
only showing top 1 row
```

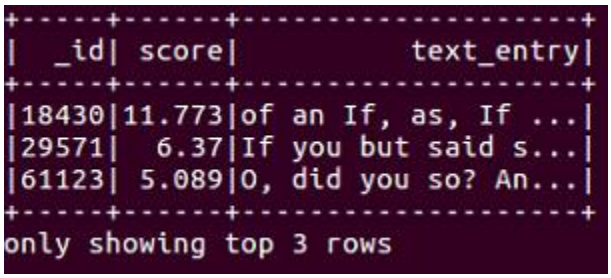
```
+-----+-----+--------------------+
|  _id|score|          text_entry|
+-----+-----+--------------------+
|68413|3.593|And so far am I g...|
|11154|3.593|So do I wish the ...|
|51283|2.764|so, so, so. Well ...|
+-----+-----+--------------------+
only showing top 3 rows
```



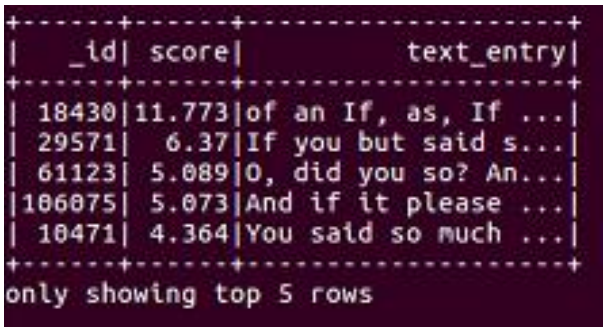
```
+------+-----+--------------------+
|   _id|score|          text_entry|
+------+-----+--------------------+
| 68413|3.593|And so far am I g...|
| 11154|3.593|So do I wish the ...|
| 51283|2.764|so, so, so. Well ...|
| 96897|2.671|That brought her ...|
|110732|2.671|nothing, let him ...|
+------+-----+--------------------+
only showing top 5 rows
```



```
+-----+------+--------------------+
|  _id| score|          text_entry|
+-----+------+--------------------+
|18430|11.773|of an If, as, If ...|
+-----+------+--------------------+
only showing top 1 row
```



```
+-----+------+--------------------+
|  _id| score|          text_entry|
+-----+------+--------------------+
|18430|11.773|of an If, as, If ...|
|29571|  6.37|If you but said s...|
|61123| 5.089|O, did you so? An...|
+-----+------+--------------------+
only showing top 3 rows
```



```
+------+------+--------------------+
|   _id| score|          text_entry|
+------+------+--------------------+
| 18430|11.773|of an If, as, If ...|
| 29571|  6.37|If you but said s...|
| 61123| 5.089|O, did you so? An...|
|106075| 5.073|And if it please ...|
| 10471| 4.364|You said so much ...|
+------+------+--------------------+
only showing top 5 rows
```