DESIGN AND IMPLEMENTATION OF
DOMAIN SPECIFIC INFORMATION RETRIEVAL SYSTEM
https://github.com/SupremeFlash

Problem Definition:

- Implementation of a text based information retrieval system
- A domain specific search engine
- The searching module is built according to the Vector Space model of Information Retrieval.
- The program builds the index of all the files, then retrieves all the documents relevant to queries input by user.
- This program performs ranked retrieval using the tf-idf scoring model.

The indexer program needs to be run just once for a file system.
The runner program needs to be run every time a query needs to be processed.

MODULES:

CommonExtractor.py:

I have extracted the .txt files from the .html files using BeautifulSoup module

Indexer.py:

Stemming using porter stemmer
Removed stop words and repetition
Convert to lower case
Normalize using regular expressions
Use word_tokenize module from NLTKpackage
Create postingslist, tf, idf indexes
Write them to text files
Calculate running time

Runner.py:

Get queries as input
Load the tf, idf indexes written by Indexer.py
Calculate the idf scores of the query terms
Calculate the tf scores for each query and doc
Get the final tf-idf scores for all docs
Sort the docIDs in the descending order of the scores
Retrieve the top 10 relevant documents
Display and write the docnames to a file as well
Calculate Precision and Recall
Calculate running time

PSEUDOCODE:

Simplified pseudocode of some of the important functions:

traversethroughdocs():

Call tokenizedoc()
Call insertinpostingslist()

tokenizedoc():

Split the doc into lines, the lines into words, and process each line
Line by Line:
Convert to lower case, normalize using regular expressions, use word_tokenize module from NLTKpackage , call cleantokensofeachdoc()

cleantokensofeachdoc():

Perform stemming, removing stop words, removing repetition, creating a dictionary named TFHelper and their counts, in that doc,

```
#basically its a single column in the tf matrix
addTotf()
```

insertinpostingslist():

```
if postingslist already exists:
    Loop through the tokens in the doc and the tokens in the postingslist
        If they match:
        Append the docID to that element in the postingslist
    Else if postingslist needs to be created:
      Loop through the tokens in the doc
        Just insert the token and the corresponding docID
return postingslist
```

addTotf():

```
newterm = 1
Loop through tokens in TFHelper
    Loop through terms in the tf index
  if they match
        Not a new term
        Append the count of the token in that doc to that term element in the tf index
    if it is a new term in the tf index:
        Insert the count of tokens in that doc to the tf index
      Loop through the terms in tf index
        If no terms for this doc (check by comparing the length of the row)
          Append a 0
```

createIDF():

```
Loop through the terms in the postingslist
    Set idf value of that term to the length of the postingslist
```

Pickle: Writing to files

Indexer.py writes the postingslist, tf, idf indexes in .txt files, for the reference of the runner.py program later.

An efficient way to write the text files is to dump the indexes as objects, using pickle module.

But if the user wishes to see or check the indexes, the object cannot be displayed suitably. Hecne, a copy of “human - readable” files have also bee written, for easy reference.

Runner.py:

cleanqueries():

Split the queries into terms

Convert to lower case, normalize using regular expressions, use word_tokenize module from NLTKpackage, stemming, removing stopwords, removing repetition

computeidf():

```
If the query term is in the idf index
    Log(N/idf[term])
Else
    0
```

computetf():

```
If query term is in the tf index
    If tf for that query and docID is 0
        Return 0
    Else
        Log(1+tfvalue)
Else
    Return 0
```

computescores():

```
Doc by doc
Query: term by term
If number of query terms>1
    Compute idf
Else
    Idfscore = 1
Computetf()
Update score <- score + tfscore + idfscore
Update to scores dictionary
```

ranking():

Sort the scores dictionary in the descending order of the values

Find the number of docs to be retrieved by counting the number of non-zero scores in the ranks dictionary.

If relevant docs>10
    Retrieve 10 docs
Else retrieve all relevant docs
getdocNames()

getdocNames():
Traverse through the ExtractedText folder, and find the name of the files correspondgin to the docIDs, by counting appropraitely

computePR():
Precision = number of /N

Formulae:

Precision = Fraction of documents that are relevant to user's information.
Recall = Fraction of relevant documents in collection that are retrieved.

MY DATASET:

I have used the corpus 'Wiki Small' from:
the book "Search Engines: Information Retrieval in Practice"'s webpage:
http://www.search-engines-book.com/collections/
The corpus was created from a snapshot of the English Wikipedia downloaded from static.wikipedia.org in September 2008.
The corpus contains 6043 articles as .html files.
which I have extracted as .txt files.
The .html files are categorized according to each consecutive letter of their filenames, but in my ExtractedText folder, I have removed the categorization and just collected all the files together.

Domain I have used : Wikipedia Articles

RUNNING TIME :

CommonExtractor.py : approx 217 seconds (for 6043 articles)
Indexer.py : approx 443 seconds
runner.py : <1s, varies for each query

PRECISION : varies for each query, and is displayed at the end of the program
RECALL : varies for each query, and is displayed at the end of the program

DATA STRUCTURES USED:

A] postingslist:

-> dictionary of string to list of integers

-> dictionary of token to list of docs containing the token

B] tf: The term frequency index

-> dictionary of string to list of integers

-> dictionary of token to frequency of occurrence of that term in that doc

C] idf: The inverse document frequency index

-> dictionary of string to int

-> dictionary of token to the frequency of occurrence of that token in the corpus, that is, the number of documents the token appears in, in the corpus

D] TFhelper:

-> dictionary of strings to integers

-> dictionary of tokens and their counts in each doc, basically its a single column in the tf matrix

E] index:

-> any dictionary or list, to be written to a text file

F] scores:

-> dictionary of integers to doubles

-> dictionary of docIDs to their tf-idf scores

G] ranks:

-> an ordered dictionary of integers to doubles

-> an ordered dictionary of docIDs to their tf-idf scores sorted in the descending order of their scores

stopwords, tokensindoc, lines, tokensinline, queries are lists of strings used to handle the tokens.