Information Retrieval

Blockchain Search Engine Detailed Report

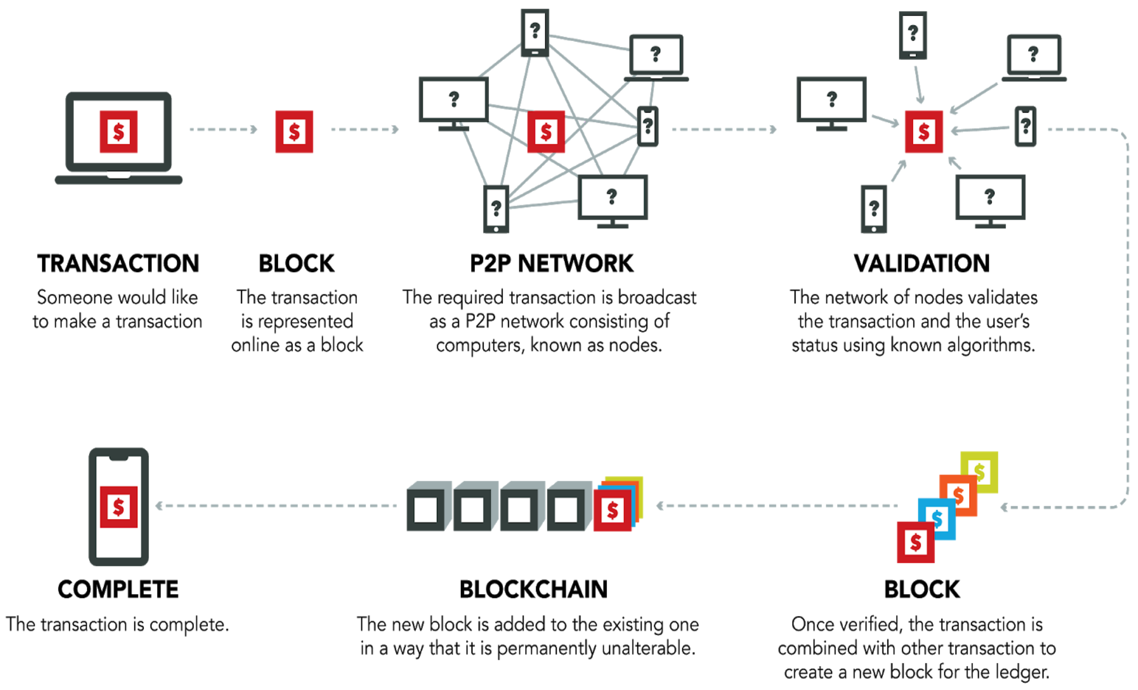
Team

Nethish R (16PT21)
Nithish S (16PT24)
GitHub: https://github.com/nethish/SearchEngine

Blockchain

Blockchain, sometimes referred to as Distributed Ledger Technology (DLT), makes the history of any digital asset unalterable and transparent through the use of decentralization and cryptographic hashing. It is a revolutionary technology that reduces risk and keeps track of all the transactions in an organization.

A blockchain is an ever-growing list of records called blocks which are linked using cryptography. Cryptography is a process which encrypts and secures data communication to prevent third-parties from reading private messages. Blockchain technology is most commonly used by cryptocurrencies. Once the data has been recorded in a place, it will not be changed. It works just like a digital notary with timestamps to avoid tampering of information.


TRANSACTION
Someone would like to make a transaction
BLOCK
The transaction is represented online as a block
P2P NETWORK
The required transaction is broadcast as a P2P network consisting of computers, known as nodes.
VALIDATION
The network of nodes validates the transaction and the user's status using known algorithms.
BLOCK
Once verified, the transaction is combined with other transaction to create a new block for the ledger.
BLOCKCHAIN
The new block is added to the existing one in a way that it is permanently unalterable.
COMPLETE
The transaction is complete.

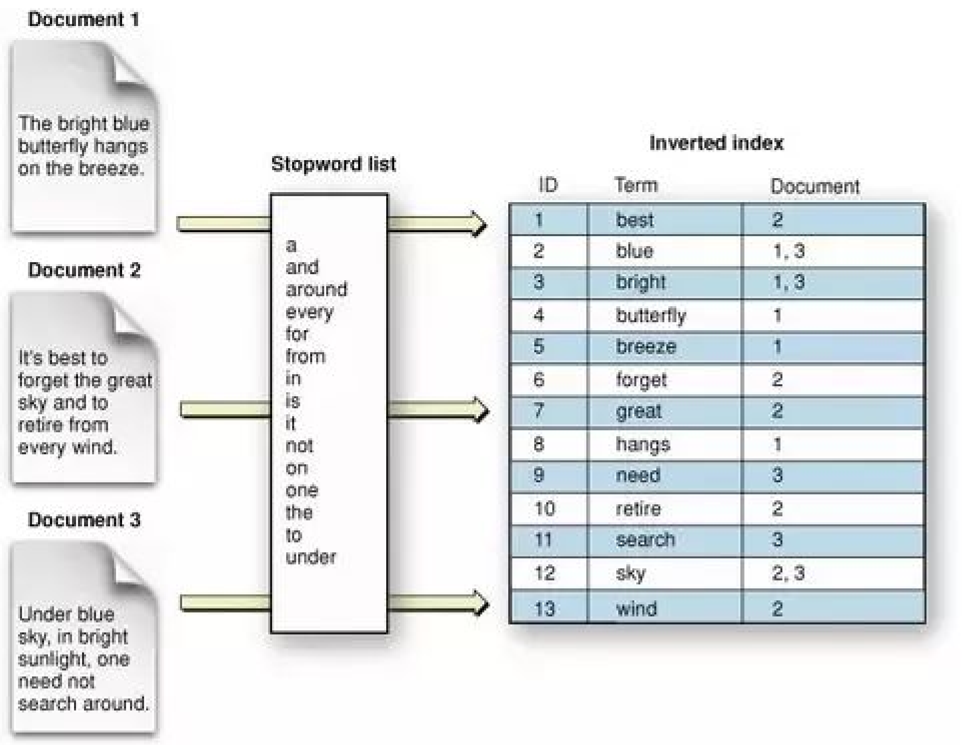
Blockchain Search Engine

There are numerous applications of blockchain and it's still being evolved. The aim is to build a search engine to investigate the current state of blockchain technology and its applications and how this technology can revolutionize modern business.

There are many theories and research papers published on IEEE(https://www.ieee.org/), Springer(https://www.springer.com/in) and in many more websites. If one wants to build an application or a model that uses blockchain technology they would have to research this area a bit to enhance the use of this technology. With its wide range of applications in the field of Cryptocurrencies, Financial services, Supply chain, Video games, Energy trading etc, many research papers are also being published in other fields too. It is very likely that the same topic has been researched and published on the internet. So building a blockchain search engine dedicated to search for the papers of the area of interest would be helpful to organizations that want to use this technology in their own way.

Indexing - Inverted Index

An inverted index (also referred to as a postings file or inverted file) is a database index storing a mapping from content, such as words or numbers, to its locations in a table. The inverted index is mainly used in search engines. The goal is to find all documents that have a particular word in it. The inverted index helps to find these kinds of queries. Building requires scraping the document, removing stop words and indexing. Exposing the data to preprocessing techniques like stemming would also increase accuracy.



Document 1: The bright blue butterfly hangs on the breeze.

Document 2: It's best to forget the great sky and to retire from every wind.

Document 3: Under blue sky, in bright sunlight, one need not search around.

Stopword list: a, and, around, every, for, from, in, is, it, not, on, one, the, to, under

ID	Term	Document
1	best	2
2	blue	1, 3
3	bright	1, 3
4	butterfly	1
5	breeze	1
6	forget	2
7	great	2
8	hangs	1
9	need	3
10	retire	2
11	search	3
12	sky	2, 3
13	wind	2

The Vector Space Model

Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers (such as index terms). It is used in information filtering, information retrieval, indexing and relevancy rankings.

The Term Space

The core of a Vector Space Engine is the Term Space itself. The idea behind it is to create a Vector Space where each dimension is represented by a term. This space can grow in dimension every time a new keyword is added. In a vector-based retrieval model, each document is represented by a vector d = (d1, d2, ..., dn) where each component di is a real number indicating the degree of importance of term ti in describing document d.

Once, all documents are represented within the common term space, the relevance between them can be rated according to various rating procedures. Before moving to document rating and term weighting, an evaluation of the presented model in respect to its distribution capabilities is necessary.

Binary weighting will not be sufficient for a sophisticated search engine. It completely ignores important information like term frequency or document length. For this reason, term weights are assigned to the vector-elements.

Stop Words

Words such as articles and some verbs are usually considered stop words because they don't help us to find the context or the true meaning of a sentence. Since it does not help in indexing or searching, removing it helps in reducing the dictionary size and in some cases also improves the performance of the search engine. The nltk library provides functions to remove such words in a given language.

Lemmatization

Lemmatization is the process of converting the given word to its root word. For example converting rocks to rock. It is a helpful preprocess technique that helps in reducing the dictionary size and term space and also captures the synonyms. Lemmatizers require a lot more knowledge about the structure of a language, it’s a much more intensive process than just trying to set up a heuristic stemming algorithm.

In general, lemmatization offers better precision than stemming, but at the expense of recall. As we've seen, stemming and lemmatization are effective techniques to expand recall, with lemmatization giving up some of that recall to increase precision.

TF-IDF

The term frequency of a word in a document. There are several ways of calculating this frequency, with the simplest being a raw count of instances a word appears in a document.

Then, there are ways to adjust the frequency, by length of a document, or by the raw frequency of the most frequent word in a document.

The inverse document frequency of the word across a set of documents. This means, how common or rare a word is in the entire document set. The closer it is to 0, the more common a word is. This metric can be calculated by taking the total number of documents, dividing it by the number of documents that contain a word, and calculating the logarithm.

So, if the word is very common and appears in many documents, this number will approach 0. Otherwise, it will approach 1.

$$idf(t, D) = log\left(\frac{N}{count\ (d \in D : t \in d)} \right)$$

$$w_{ik} = tf_{ik} * ld(\frac{N}{n_k} + 1).$$

$tf_{i,\,k}$ = frequency of term Tk in document Di
idf_k = inverse document frequency of term Tk in collection C.

TF-IDF normalization (Improvised)

One drawback with tf-idf is that it rates the longer documents higher than the shorter ones. Out of this reason, term weights are usually normalized to an in-terval between 0 and 1, so the total number of occurrences within one document does not matter anymore or it has only very little effect. Binary weighting like presented in the previous section will not be sufficient for a sophisticated search engine. It completely ignores important information like term frequency or document length. For this reason, term weights are assigned to the vector-elements. A common method to assign term weights is to store the inverse document frequency of a document as the vector element. So to normalize it the following formula is used to improvise it

$$w_{ik} = \frac{tf_{ik} * ld(\frac{N}{n_k})}{\sqrt{\sum_{k=1}^{t}(tf_{ik})^2[ld(\frac{N}{n_k})]^2}}$$

Where **ld** is binary logarithm. Binary logarithm is used here because the $\log_{10}$ sometimes overy dampens the frequency.

Normalization is generally used in two ways: first, to prevent bias in term frequency from terms in shorter or longer documents; second, to calculate each term's idf value. There are many ways to normalize the score but re-weighting the tfidf is also important.

Latent Semantic Analysis

LSI takes documents that are semantically similar (= talk about the same topics), but are not similar in the vector space (because they use different words) and re-represents them in a reduced vector space in which they have higher similarity. Similar terms map to similar locations in low dimensional space. Noise reduction by dimension reduction. Latent Semantic Analysis is an efficient way of analysing the text and finding the hidden topics by understanding the context of the text. Latent Semantic Analysis(LSA) is used to find the hidden topics represented by the document or text. This hidden topics then are used for clustering the similar documents together.

Then each document is semantically analyzed and reduced using the algorithm. These vector representation of the document is then processed and stored in an index to process the queries.

Querying and Ranking

The documents are first used to construct a document term matrix. These values are normalized and tf-idf scores are calculated. Now each of the documents is represented as vectors with normalized tf-idf values for terms which are reduced using lsi mode. The document representation is extracted from the lsi model.

The structure of a query is a simple string consisting of space separated words. This query undergoes a construction and restructuring process corresponding to the tf-idf matrix. Then this restructured query is used for Ranking the documents. The words that are not present in the copus is completely ignored.

Cosine similarity metric is used for finding the similar documents and ranking.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

The lowest similarity is 0 which says there are no similarities between documents and highest is 1 which indicates both the documents are the same. Basically higher the value more similar the documents are. A simple search returns most similar 5 links.

Caching

Computing the similarities between documents and ranking them takes time when the collection time is huge. Also when a new document is added it has to re compute the entire lsi matrix for semantic analysis and consideration. These pose hard time constraint on the search engine. Hence caching must be used to reduce the latency between searches and rebuilding the indices.

Index Caching: This step involves caching of indices so that it can be loaded and used everytime the search engine is shut down and restarted. This eliminates the time needed to fetching the documents, parsing it, converting it to a bag of words and lemmatizing it. This level of caching also helps when search queries are issued to compute the frequencies of the words in the documents.

Query Caching: This caches queries to improve the performance of the system. Whenever a query is issued it checks with the cache if it is present. If the query is already in the cache it returns it otherwise it recomputes the query and similarities and puts it into the cache for future query processing.

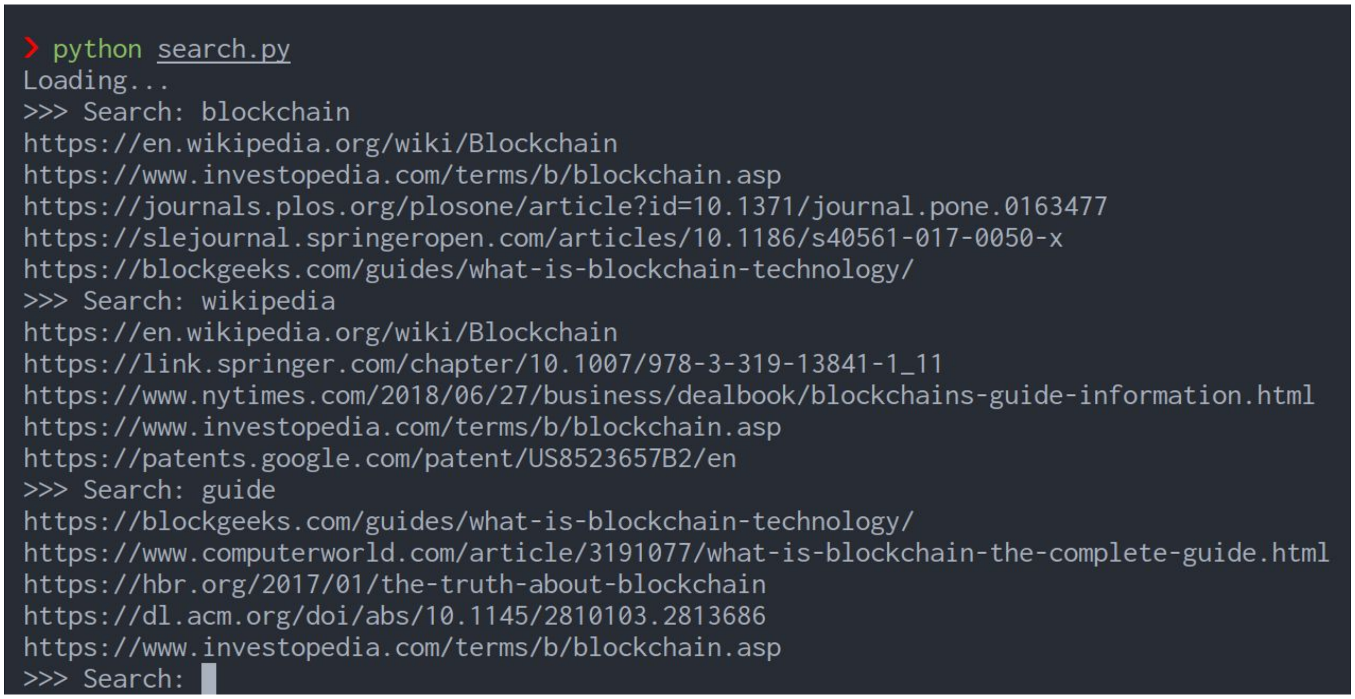
Performance: The performance has been measured by issuing the same queries twice. The time it takes to process queries becomes less.

```
Search: blockchain technologies
Search time:  0.41754698753356934 seconds
https://www.computerworld.com/article/3191077/what-is-blockchain-the-comple
te-guide.html
https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/ey
al
https://dl.acm.org/doi/abs/10.1145/2504730.2504747
https://www.nytimes.com/2018/06/27/business/dealbook/blockchains-guide-info
rmation.html
https://en.wikipedia.org/wiki/Blockchain
>>> Search: blockchain technologies
Search time:  0.07686352729797363 seconds
https://www.computerworld.com/article/3191077/what-is-blockchain-the-comple
te-guide.html
https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/ey
al
https://dl.acm.org/doi/abs/10.1145/2504730.2504747
https://www.nytimes.com/2018/06/27/business/dealbook/blockchains-guide-info
rmation.html
https://en.wikipedia.org/wiki/Blockchain
```

For the same search queries it reduces the time needed by 6 times.

Example searches

The application has been tested with 40 block chain websites, research papers and some other pages that provide information about blockchain. Currently the search engine outputs 5 links which can be tuned to get more relevant results.



```
❯ python search.py
Loading...
>>> Search: blockchain
https://en.wikipedia.org/wiki/Blockchain
https://www.investopedia.com/terms/b/blockchain.asp
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0163477
https://slejournal.springeropen.com/articles/10.1186/s40561-017-0050-x
https://blockgeeks.com/guides/what-is-blockchain-technology/
>>> Search: wikipedia
https://en.wikipedia.org/wiki/Blockchain
https://link.springer.com/chapter/10.1007/978-3-319-13841-1_11
https://www.nytimes.com/2018/06/27/business/dealbook/blockchains-guide-information.html
https://www.investopedia.com/terms/b/blockchain.asp
https://patents.google.com/patent/US8523657B2/en
>>> Search: guide
https://blockgeeks.com/guides/what-is-blockchain-technology/
https://www.computerworld.com/article/3191077/what-is-blockchain-the-complete-guide.html
https://hbr.org/2017/01/the-truth-about-blockchain
https://dl.acm.org/doi/abs/10.1145/2810103.2813686
https://www.investopedia.com/terms/b/blockchain.asp
>>> Search:
```

```
❯ python search.py
Loading...
>>> Search: blockchain
https://en.wikipedia.org/wiki/Blockchain
https://www.investopedia.com/terms/b/blockchain.asp
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0163477
https://slejournal.springeropen.com/articles/10.1186/s40561-017-0050-x
https://blockgeeks.com/guides/what-is-blockchain-technology/
>>> Search: wikipedia
https://en.wikipedia.org/wiki/Blockchain
https://link.springer.com/chapter/10.1007/978-3-319-13841-1_11
https://www.nytimes.com/2018/06/27/business/dealbook/blockchains-guide-info
rmation.html
https://www.investopedia.com/terms/b/blockchain.asp
https://patents.google.com/patent/US8523657B2/en
>>> Search: guide
https://blockgeeks.com/guides/what-is-blockchain-technology/
https://www.computerworld.com/article/3191077/what-is-blockchain-the-comple
te-guide.html
https://hbr.org/2017/01/the-truth-about-blockchain
https://dl.acm.org/doi/abs/10.1145/2810103.2813686
https://www.investopedia.com/terms/b/blockchain.asp
```

The red terms are the search terms and the green are the relevant links. The links contain some relevant information about the document. We could see some of the links matching with the content and the query. Click the relevant link to check out the website.

Libraries used

1. requests - for requesting and fetch web pages
2. BeautifulSoup - To extract text based on tags. This parses the fetched web page and return it as a string which is convenient for processing
3. nltk.stopwords - To remove stop words
4. WordNetLemmatizer - To reduce the terms to the root word.
5. pickle - To persist the data extracted in the main memory for further processing.

Code

scrape.py

```
import requests
import bs4 as bs
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

stop_words = set(stopwords.words('english'))

def get_text(url):
    try:
        response = requests.get(url);
    except:
        print("Couldn't get:", url)
        return ''
    if response.status_code != 200:
        return ''
    text = bs.BeautifulSoup(response.content, 'html.parser').text
    return text

def tokenize(text):
    text = text.strip('\n\r')
    tokens = re.split('\W+', text)
    token_list = []
    for i in tokens:
        if not i or len(i) > 20:
            continue
        token_list.append(i.lower())
    # tokens = []
    # for line in lines:
        # toks = line.split(' ')
        # for t in toks:
            # if not t:
                # continue
            # tokens.append(t)
    token_list = [token for token in token_list if token not in stop_words]
    return token_list

def lemmatize(tokens):
    lemmatizer = WordNetLemmatizer()
    for i in range(len(tokens)):
        tokens[i] = lemmatizer.lemmatize(tokens[i])
    return tokens

def get_tokens(url):
    text = get_text(url)
    tokens = tokenize(text)
    tokens = lemmatize(tokens)
    return tokens
```

Functions:

get_text - Extracts the text information from the provided link
tokenize - Convert the given string into list of tokens. Remove stop words
lemmatize - Convert each token to its root word

collection.py

```
import pickle
from collections import defaultdict
from scrape import get_tokens, get_text
from math import log

def getDD():
    return defaultdict(int)

class Collection:
    def __init__(self):
        self.urls = {}
        self.corpus = set()
        self.postings = defaultdict(getDD)
        self.tdf = defaultdict(getDD)
        self.terms = dict()
        self.vocabulary = 0
        self.size = 0

    def add_document(self, url):
        if url in self.urls:
            return
        tokens = get_tokens(url)
        if len(tokens) == 0:
            return
        DOC_ID = len(self.urls) + 1
        self.urls[DOC_ID] = url
        self.corpus.add(url)
        self.size += 1
        for tok in tokens:
            if tok not in self.terms:
                self.vocabulary += 1
                self.terms[tok] = self.vocabulary
            self.tdf[DOC_ID][tok] += 1
            doc_list = self.postings[tok]
            doc_list[DOC_ID] += 1
            self.postings[tok] = doc_list

    def get_tdf(self, doc, term):
        return self.tdf[doc][term]

    def get_dft(self, term):
        return len(self.postings[term])

    def get_tfidf(self, doc, term):
        tdf = self.get_tdf(doc, term)
        dft = self.get_dft(term)
        return (log(tdf + 2)) * log(self.size / (1 +dft))

    def get_url(self, id):
        return self.urls[id]

    # def get_postings(self):
    #     for i in self.postings:
    #         print('Word:', i)
    #         for j in self.postings[i]:
    #             print('Doc: %d, Freq: %d'%(j, self.postings[i][j]), end =
' | ')
```

```
    #         print('\n-------------------------------')
    #     return self.postings

if __name__ == "__main__":
    links = open("./Links.txt", 'r').read().split('\n')
    collection = Collection()
    for link in links:
        collection.add_document(link)
```

The Collection class keeps track of the given urls and updates the index when ever new document is processed. It stores the index in the main memory. It also keeps track of term frequency in a document and a postings list. These two indices helps in calculating the tfidf efficiently whenever needed.

search.py

```
from collection import Collection
import re, os
import pickle
from math import log, log2
import lsi

class Search:
  def __init__(self, collection):
    self.collection = collection
    self.terms = []
    self.search_limit = 5
    self.set_collection(collection)

  def set_collection(self, collection):
    self.collection = collection
    self.terms = []
    for term in self.collection.terms:
      self.terms.append(term)
    self.update_lsi()

  def update_lsi(self):
    matrix = []
    for doc in range(self.collection.size):
      matrix.append([])
      for t in self.terms:
        matrix[-1].append(self.normalize_tfidf(doc, t))
    self.lsi = lsi.lsi_matrix(matrix)
  def get_collection(self):
    return self.collection

  def cosine(self, a, b):
    n = len(a)
    dot = 0
    norm_a = norm_b = 0
    for i in range(n):
      dot += a[i] * b[i]
      norm_a += a[i] ** 2
      norm_b += b[i] ** 2
    if not norm_a or not norm_b:
      return 0
    cos = dot / (norm_a * norm_b) ** 0.5
    return cos

  def structure_query(self, query):
    terms = re.split('\W+', query)
    for i in range(len(terms)):
      terms[i] = terms[i].lower()
    query = []
    for t in self.terms:
      cnt = terms.count(t) + 1
      if not cnt:
        query.append(0)
        continue
      inv = log2((self.collection.size + 1) / (2 +
self.collection.get_dft(t)))
      res = log(log2(cnt + 1)) * inv
      res /= ((cnt ** 2) * inv ** 2) ** ( 1 / 2)
      query.append(res)
    return query

  def get_lsi_doc(self, doc):
    return list(self.lsi[doc - 1])

  def search(self, query):
    # result = []
    # terms = re.split('\W+', query)
    # query = []
    # for t in self.terms:
    #   cnt = terms.count(t)
    #   if not cnt:
    #     query.append(0)
    #     continue
    #   query.append(log(1 + log(cnt)) * log((self.collection.size + 1) /
(1 + self.collection.get_dft(t))))
    query = self.structure_query(query)

    if not query:
      return ["No match found"]

    result = []
    for doc in range(1, len(self.collection.urls) + 1):
      doc_vector = self.get_lsi_doc(doc)
      # for t in self.terms:
      #   doc_vector.append(self.collection.get_tfidf(doc, t))
#self.collection.tdf[doc][t])
      result.append((self.cosine(doc_vector, query), doc))

    result.sort()
    search_results = []
    for i in range(min(self.search_limit, len(result))):
      search_results.append(self.collection.get_url(result[i][1]))
    # print(result)
    return search_results

  def normalize_tfidf(self, doc, term):
    ntfidf = self.collection.get_tfidf(doc, term)
    tf = self.collection.get_tdf(doc, term) + 1
    df = self.collection.get_dft(term)
    size = self.collection.size
    ntfidf /= (tf ** 2 * log2(2 + size / (df + 1))) ** (1 / 2)
    return ntfidf

  def construct_doc(self, doc):
    doc_vector = []
    for t in self.terms:
      doc_vector.append(self.normalize_tfidf(doc, t))
    return doc_vector

  def update_collection(self, links):
    updated = False
    for link in links:
      if link not in self.collection.corpus:
        updated = True
        print("Updated: ", self.collection.size)
        self.collection.add_document(link)
    if not updated:
      return
    self.set_collection(self.collection)
    collection_loader = CollectionLoader()
    collection_loader.dump(self.collection)

class CollectionLoader:
  def __init__(self):
    pass

  def dump(self, collection, file='index.pickle'):
    with open(file, 'wb') as handle:
      pickle.dump(collection, handle)

  def load(self, file='index.pickle'):
    file = open('index.pickle', 'rb')
    collection = pickle.load(file)
    file.close()
    return collection

if __name__ == "__main__":
    links = open("./Links.txt", 'r').read().split('\n')
    collection_loader = CollectionLoader()
    if os.path.isfile('index.pickle'):
      print('Loading...')
      collection = collection_loader.load()
    else:
      print('Scraping...')
      collection = Collection()
      collection_loader.dump(collection)
    search = Search(collection)
    search.update_collection(links)
    # print(collection.terms)
    query = input('>>> Search: ')
    while query != 'exit':
      result = search.search(query)
      for i in result:
        print(i)
      query = input('>>> Search: ')
```

This file contains the CollectionLoader and Search class. The CollectionLoader class persists the index in the secondary memory so that it need not be calculated every time the engine is rerun.

The Search class also updates the lsi indices whenever a new document is added to the library. The whole lsi matrix is recomputed and used since new words in the library can potentially change the weight of the words.

The search class functionalities are to keep track of the collection and process the search query based on the heuristic provided. It structures the query and normalizes the weights for computing cosine similarity.

lsi.py

```
import numpy as np

def lsi_matrix(matrix, K=5):
  U, s, VT = np.linalg.svd(matrix)
  reduced_matrix = np.dot(U[:, :K], np.dot(np.diag(s[:K]), VT[:K, :]))
  return reduced_matrix
```

References

1. https://www.researchgate.net/publication/4223307_A_vector_space_search_engine_for_Web_services
2. Research on Text Classification Based on Improved TF-IDF Algorithm - Atlantis press
3. An Integrated and Improved Approach to Terms Weighting in Text Classification - IJCSI International Journal of Computer Science Issues