# Simple Search Engine

**Soumya Chiday(U00864145)**
**Ram Somesh(U00873934)**

## Index.py:

```
'''

Index structure:

    The Index class contains a list of IndexItems, stored in a dictionary type for easier access

    each IndexItem contains the term and a set of PostingItems

    each PostingItem contains a document ID and a list of positions that the term occurs

'''
from __future__ import print_function
import util
from cran import *
import doc
import sys #File IO
import re #Regex
import nltk
import collections
from nltk.tokenize import RegexpTokenizer
import codecs
import string
import pickle


class Posting:
    def __init__(self, docID):
        self.docID = docID
        self.positions = []

    def append(self, pos):
        self.positions.extend(pos)

    def sort(self):
        ''' sort positions'''
        self.positions.sort()

    def merge(self, positions):
```

```
        self.positions.extend(positions)

    def term_freq(self):
        ''' return the term frequency in the document'''
        #ToDo
            tf=len(self.positions)
            return tf


class IndexItem:
    def __init__(self, term):
        self.term = term
        self.posting = {} #postings are stored in a python dict for easier index building
        self.sorted_postings= [] # may sort them by docID for easier query processing

    def add(self, docid, pos):
        ''' add a posting'''
        if not self.posting.has_key(docid):
            self.posting[docid] = Posting(docid)
        self.posting[docid].append(pos)

    def sort(self):
        ''' sort by document ID for more efficient merging. For each document also sort the
positions'''
        # ToDo
            for docid in self.posting.keys():
                self.posting[docid].sort()
                self.sorted_postings.extend(self.posting[docid].positions)

class InvertedIndex:

    def __init__(self):
        self.items = {} # list of IndexItems
        self.nDocs = 0  # the number of indexed documents
        self.endmost={}

    def tokenization(body):
        body = re.sub("[^a-z0-9]+", " ", body)
        tokens = nltk.tokenize.word_tokenize(body)
        return tokens

    def indexDoc(self, doc): # indexing a Document object
        # ToDo: indexing only title and body; use some functions defined in util.py
        # (1) convert to lower cases,
        # (2) remove stopwords,
        # (3) stemming
```

```
    reducedList=[] #created an empty dictionary
    docBody=doc.body #body of the document is assigned to docBody
    #print(doc.docID)

    #reduced=nltk.word_tokenize(docBody) drops the non-alphanumeric terms
    tokenizer = RegexpTokenizer(r'\w+')
    reduced=tokenizer.tokenize(docBody)
    #docBody = re.sub("[^a-z0-9]+", " ", docBody)
    #reduced=nltk.tokenize.word_tokenize(docBody)
    #i=[i.lower() for i in reduced]
    #all terms are coverted to lower cases
    #print (i)
    for words in reduced:
    #reduced terms are passed through stopwords and stemming in util
        if util.isStopWord(words):
          reducedList.append(util.stemming(words))
    #normalized terms are stored in reducedList Dictionary
    #print (reducedList)

posDict={} #Dictionary is created to store (terms, positions) in documents
    for position, word in enumerate(reducedList):
    #{term1:[pos1, pos3..], term2:[..], ..} dictionary is created
        if word in posDict.keys():
            posDict[word].append(position)
        else:
                posDict[word]=[position]
    #print(posDict)

wordKeys=[] #keys in posDict are stored
wordPos=[] #values in posDict are stored
wordKeys=posDict.keys()
#print(wordKeys)
for i in range(0,len(posDict),1):
    #appends the values of the corresponding key into wordPos
    wordPos.append(posDict[wordKeys[i]])

#print(wordPos)
i=0

    for word in posDict:
    #to assign the positions of a word to its respective docID
    docdic={}
    #pos=wordPos[i]
    if word in self.items.keys():
        #if word is already in the item.keys(), then just add the docID
        pos=posDict[word]
```

```
            self.items[word].add(int(doc.docID),pos)
            for i in range(0, 1401, 1):
                  #for every doc, add the docId and positions to the endmost dictionary
               if i in self.items[word].posting.keys():
                        docdic[i]=self.items[word].posting[i].positions
               self.endmost[word]=docdic
         else:
            #if word isnt present in items.keys(), create an index item, then add docId to positions
            obj_Index_item=IndexItem(word)
            pos=posDict[word]
            obj_Index_item.add(int(doc.docID),pos)
            self.items[word] =obj_Index_item
            for i in range(0,1401,1):
               #for every doc, add the docId and positions to the endmost dictionary
               if i in self.items[word].posting.keys():
                  docdic[i]=self.items[word].posting[i].positions
            self.endmost[word]=docdic

            i=i+1
         print(self.endmost)
      #for key in sorted(self.endmost):
         #srt=self.endmost.keys()
         #srt.sort()
         #for s in srt:
            #print((s, self.endmost[s]))
         print(len(self.endmost))


def save(self, filename):

      # ToDo: using your preferred method to serialize/deserialize the index
   savePickle = open(filename, "wb")
   pickle.dump(self.endmost, savePickle)
      savePickle.close()
      #load(pickleFile)


def load(self, filename):

   # ToDo
   loadPickle=open(filename, "rb")
      self.endmost=pickle.load(loadPickle)
      return self.endmost
      #print(loadedDict)
```

```
    def sort(self):
        ''' sort all posting lists by docID'''
        #dict=collection.OrderedDict(sorted(dict()))
            #return dict

    def find(self, term):
        return self.items[term]


    # more methods if needed


def test():
    ''' test your code thoroughly. put the testing cases here'''
    print ('Pass')

def indexingCranfield(filee, filename):
    #ToDo: indexing the Cranfield dataset and save the index to a file
    # command line usage: "python index.py cran.all index_file"
    # the index is saved to index_file

    #cranObj=CranFile()
    iiObj=InvertedIndex()
    #object for inverted index is created
    cf = CranFile (filee)
    #cran.all is uploaded from an cran.py object

    for doc in cf.docs:
    #for the documents in corpus, all the terms are sorted
            iiObj.indexDoc(doc)
    iiObj.sort()
    #terms are sorted and saved in as a pickle file
    iiObj.save(filename)
    #print(rtn)

    print ('Done')

if __name__ == '__main__':
    #test()

    corpusFile=str(sys.argv[1])
    pickleFile=str(sys.argv[2])
    indexingCranfield(corpusFile, pickleFile)
```

## Query.py:

```
'''
query processing

'''
from nltk.tokenize import RegexpTokenizer
from norvig_spell import correction
import util
from numpy import dot
from numpy.linalg import norm
from operator import itemgetter
from cranqry import loadCranQry
from index import *
from index import *
from time import sleep
import sys

class QueryProcessor:

    def __init__(self, query, index):
        ''' index is the inverted index; collection is the document collection'''
        self.raw_query = query
        self.index=index
        #self.docs = collection
        self.query=[]

    def preprocessing(self):
        ''' apply the same preprocessing steps used by indexing,
            also use the provided spelling corrector. Note that
            spelling corrector should be applied before stopword
            removal and stemming (why?)'''
            qbody=self.raw_query
        #ToDo: return a list of terms
        #cqObj=CranFile('query.text')
            #qbody=cqObj.body
            print qbody

            qbody=re.sub("[^a-z0-9]+", " ", qbody)
            reduced=nltk.tokenize.word_tokenize(qbody)
```

```
            for words in reduced:
            #reduced terms are passed through stopwords and stemming in util
                if util.isStopWord(words):
                    self.query.append(util.stemming(words))
            #normalized terms are stored in reducedList Dictionary
            print (self.query)


        '''correctedwords=[correction(word) for word in words]
        lowercasewords = [word.lower() for word in correctedwords]
        notstopwords = []
        for word in lowercasewords:
            if not util.isStopWord(word):
                notstopwords.append(util.stemming(word))
        self.query=notstopwords'''


    def booleanQuery(self):
        ''' boolean query processing; note that a query like "A B C" is transformed to "A AND B
AND C" for retrieving posting lists and merge them'''
        #ToDo: return a list of docIDs
        #print self.query

            iiObj=InvertedIndex()
            loadedindex=iiObj.load('index_file.pickle')
        retrievedQueries=set(loadedindex[self.query[0]].posting.keys())
        #print retrievedQueries
        for term in self.query:
            #print retrievedQueries
                retrievedQueries=retrievedQueries.intersection(loadedindex[term].posting.keys())
            #print retrievedQueries


    def vectorQuery(self, k):
        ''' vector query processing, using the cosine similarity. '''
        #ToDo: return top k pairs of (docID, similarity), ranked by their cosine similarity with the
query in the descending order
        # You can use term frequency or TFIDF to construct the vectors
            tfs=[]
            #term frequncies for every term
        df=[]
            #document frequency list to store it for each term
        vecRanking={}
            #a dictionary to store the cosine similarities
        terms=self.index.items.keys()
            #the terms in dictionary for comparision
```

```
        terms=set(terms).union(self.query)
        for term in terms:
            # for every term, tf is stored in ranking dictionary by appending
            tfs.append(tf(term,self.query))
        #print sum(tfs)
        for doc in range(self.index.numofterms):
            #for every term, df is stored to calculate the idf for ranking
            df=[]
                i=0
            for term in terms:
                #tfidf of each term is stored in the dictionary
                    i=self.index.tf(term,str(doc+1))
            vecRanking[doc+1]=cos(tfs,df)
                #ranking is stored in dictionary
            topk(k,vecRanking)

def test():
    ''' test your code thoroughly. put the testing cases here'''
    print ('Pass')

def tfOfQuery(term,query): #num of times a word appearing is query is returned
    i=0
    for word in query:
        if word==term:
            i+=1
    return i

def numofterms(term, toks): #num of tokens are counted
    return toks.count(term.lower())

def tf(term, toks): #number of terms per total tokens
    return numofterms(term, toks) / float(len(toks))

def df(term, tokslist): #returns df of a term
    x = 0
    for toks in tokslist:
        if numofterms(term, toks) > 0:
            x += 1
    return x

def idf(term, tokslist): #idf is calculated by n/df
    return len(tokslist) / float(df(term, tokslist))

def tfidf(term, toks, tokslist): # tfidf is calculated  for cosine similarity
    return tf(term, toks) * idf(term, tokslist)
```

```
def cos(query,doc): #cosine similarity is determined by dot product of query and document
    return dot(query,doc)/(norm(query)*norm(doc))

def topk(k,itemsDic): #Only top K=3 are retrieved for vector model
    items=sorted(itemsDic.items())
    for i in range(k):
        print (items[i]) #op

def query(processing_algorithm,query,index): #args for command line
    ''' the main query processing program, using QueryProcessor'''

    # ToDo: the commandline usage: "echo query_string | python query.py index_file
processing_algorithm"
    # processing_algorithm: 0 for booleanQuery and 1 for vectorQuery
    # for booleanQuery, the program will print the total number of documents and the list of
docuement IDs
    # for vectorQuery, the program will output the top 3 most similar documents
    qp=QueryProcessor(query,index)
    qp.preprocessing()
    if(processing_algorithm==0):
        qp.booleanQuery()
    else:
        qp.vectorQuery(3)


if __name__ == '__main__':


    qrys =loadCranQry('query.text')
    #query.text is retrieved from loadCranQry
    index=sys.argv[1]
    #arg 1 in command line is pickle file
    qr=QueryProcessor(query, index)
    qr.preprocessing()

    alg=sys.argv[2]
    #arg 2 is 0 for bool, 1 for vector
    if (alg == '0'):
            qr.booleanQuery()
    else:
            qr.vectorQuery(3)

    query=qrys[qid].text
    query=sys.argv[3]
    #arg 3 for getting query.text
    qid=sys.argv[4]
```

```
    #arg 4 for query id
```

## Util.py:

```
'''
   utility functions for processing terms

    shared by both indexing and query processing
'''

from nltk.stem.porter import *
import re
import nltk

stemmer = nltk.PorterStemmer()

def isStopWord(word):
    ''' using the NLTK functions, return true/false'''

    # ToDo
    f=open("stopwords", 'r')
    #stopwords file is uploaded
    stopword=[]
    for words in f:
    #words are split in document, stored in a list
        wrd=words.split()
        for e in wrd:
            #each word is appended
             stopword.append(e)
    #print stopword
    if word in stopword:
            #if word is a stopword, it drops that word in reduced list
            return False
    else:
            return True
    f.close()

def stemming(word):
    ''' return the stem, using a NLTK stemmer. check the project description for installing and
using it'''

    # ToDo
```

```
st=stemmer.stem(word)
#stemmer stem all words to a root word
return st
```