Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

index.py

```
'''
Authors: Manas Gaur, Amanuel Alambo
Instructor: Dr. Keke Chen

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

'''
import util
import doc

from collections import OrderedDict,defaultdict   #for dealing with sorting
dictionaries based on keys or values---and a dictionary of lists
import math
import cran
import json
import sys

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

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

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

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
    def merge(self, positions):
        self.positions.extend(positions)

    def term_freq(self):
        ''' return the term frequency in the document'''
        #ToDo
        return len(self.positions)   #term frequency in a document is the
length of the list of positions for a document


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
        self.posting = OrderedDict(sorted(self.posting.items(), key=lambda
t:t[0]))
        for docID,pos_list in self.posting.iteritems():
            self.posting[docID] = sorted(pos_list, reverse=False)


class InvertedIndex:

    def __init__(self):
        self.items = defaultdict(list) # list of IndexItems--this
corresponds to a structure of the form
{term-1:{doc-1:[pos-1,...,pos-n]},....}
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
        self.nDocs = 0  # the number of indexed documents


    def indexDoc(self, doc): # indexing a Document object
        ''' indexing a docuemnt, using the simple SPIMI algorithm, but no
need to store blocks due to the small collection we are handling. Using
save/load the whole index instead'''

        # ToDo: indexing only title and body; use some functions defined in
util.py
        # (1) convert to lower cases,
        # (2) remove stopwords,
        # (3) stemming


        doc_tb = doc.title + doc.body   #concatenating the title and the
body as we want to index just the title and the body
        words = util.splitDoc(doc_tb)   #call to util's splitDoc method
which performs calls tokenizing, stemming, stopword removal, lowercasing
and splitting


        #pos_list = list()
        for term in list(set(words)):   #iterating through the set of words
of a document so a single word is iterated once
            term = str(term)    #conversion into a string
            term_positions = [i for i,val in enumerate(words) if val==term]
            doc_dict = {}     #dictionary to store a document's docID as
key and a term's position as value
            doc_dict[doc.docID] = term_positions    #term positions in a
document
            if term not in list(self.items.keys()):    #condition to check
presence of a key in a dict before appending a value
                self.items[term]=list()              #intializes a
dictionary for a term if the term doesn't appear as a key in the dictionary
                self.items[term].append(doc_dict)     #a dictionary of
docID as key and list of the positions of a term in a doc inserted to into
items
            else:
                self.items[term].append(doc_dict)     #if term is present
```

```
in the items dictionary the docID and term positions are appended

        self.nDocs += 1      #increments with each call to the method
        return self.items


    def sort(self):
        ''' sort all posting lists by docID'''
        #ToDo
        #sorting by posting lists by docID is handled in 'sort' method in
class 'IndexItem' (a slight change to design)

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


    def save(self, filename):
        ''' save to disk'''
        # ToDo: using your preferred method to serialize/deserialize the
index
        with open(filename, 'w') as f:
            json.dump(self.items, f, indent=4)     #json used for
serializing

    def load(self, filename):
        ''' load from disk'''
        # ToDo
        with open(filename) as f:
            indexed_file = json.load(f)
        return indexed_file

    def idf(self, term):
        ''' compute the inverted document frequency for a given term'''
        #ToDo: return the IDF of the term
        idf_score = math.log(self.nDocs, len(self.items[term]))
        return idf_score

    # more methods if needed
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
def test():
    ''' test your code thoroughly. put the testing cases here'''
    II = InvertedIndex()   #InvertedIndex class instantiated
    index_loaded = II.load('test.json')   #loading an indexed file---name
of the file passed should be the same name used in indexing and saving the
file
    print type(index_loaded['four'])     #testing for example entry 'four'
    print (index_loaded['four'][0])

    print 'Pass'

def indexingCranfield():
    #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

    #input parameters from command line
    cran_file = sys.argv[1]
    index_file = sys.argv[2]

    cf = cran.CranFile(cran_file)  #instantiation of CranFile class

    II = InvertedIndex()   #InvertedIndex class instantiated
    dump=list()       #to store list of each doc to be indexed
    for doc in cf.docs:
        dump.append(II.indexDoc(doc))

    II.save(index_file)    #saves to an output file----call to 'save'
method of 'InvertedIndex' class

    print 'Done'

if __name__ == '__main__':

    indexingCranfield()   #call to indexingCranfield method
    test()    #call to method 'test'
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

util.py

```
'''
Authors: Manas Gaur, Amanuel Alambo
Instructor: Dr. keke Chen
   utility functions for processing terms

    shared by both indexing and query processing
'''
import cran
import nltk
from nltk.tokenize import word_tokenize

from nltk.stem.snowball import SnowballStemmer


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

    # ToDo
    #returns true of a word passed is a stop word
    with open('stopwords', 'r') as f:
         stop_words = f.read()
    return word in stop_words
    #####

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

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
    # ToDo
    ##SnowballStemmer
    stemmer = SnowballStemmer("english", ignore_stopwords=True)
    return stemmer.stem(word)
    #######

#splitting a doc
def splitDoc(doc):
    tokens = word_tokenize(doc)
    words = [word for word in tokens if word.isalpha()]   #punctuation
removal
    words = [word for word in words if not isStopWord(word)]   #stopwords
removal
    words = [stemming(word) for word in words]    #word stemming
    words = [word.lower() for word in words]    #lowercasing words
    return words


###### test snippet
if __name__ == '__main__':
    print isStopWord('ffff')
    print stemming('athletes')

    docs_list = []
    cf = CranFile ('../CranfieldDataset/cran.all')   #preprocess all docs
in the collection
    for doc in cf.docs:
        docs_list.append(splitDoc(doc))

#####
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

query.py

```
'''
Authors: Manas Gaur, Amanuel Alambo
Instructor: Dr. keke Chen
query processing

'''
import util
import norvig_spell
import index
import math    #for cosine similarity
from itertools import izip
import json
from collections import defaultdict, OrderedDict
import cran

import cranqry
import numpy as np
import operator
import pandas as pd
import random
import metrics
from scipy.stats import wilcoxon
import sys
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
import random
import copy

class QueryProcessor:

    def __init__(self, query, index, collection):
        ''' index is the inverted index; collection is the document
collection'''
        self.raw_query = query
        self.index = index
        self.docs_fname = collection

        self.cf = cran.CranFile ('../CranfieldDataset/cran.all')
        self.nDocs=len(self.cf.docs)

    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?)'''

        #ToDo: return a list of terms

        corrected_terms_list = list()
        for term in self.raw_query.split(' '):   #splitting on white space
            corrected_terms_list.append(norvig_spell.correction(term))
        try:
                    corrected_terms_list.remove('gw')    #since we used the
Cranfield dataset for spelling correction, 'gw' appeared and we remove here
        except:
            pass
        corrected_terms_text = ' '.join(corrected_terms_list)

        terms = util.splitDoc(corrected_terms_text)

        return terms   #list of terms


    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

        terms = self.preprocessing()


        inv_index = self.index

        doc_list = list()    #list of docIDs for all terms
        for term in terms:
            print term
            term_doc_list = list()   #list of docIDs for a single term
            if inv_index.has_key(term):
                  for i in inv_index[term]:
                        key = ''.join(i.keys())
                        term_doc_list.append(key)


            else:
                  continue

            if len(doc_list) == 0:  #a workaround to deal with an initially
empty list---as intersection with an empty list is an empty list
                  doc_list = term_doc_list
            else:
                  doc_list = list(set(doc_list).union(set(term_doc_list)))
                  '''
              if len(doc_list) == 0:
                  doc_list = term_doc_list   #doc_list is first initialized
with a list of docIDs where the first term appears in
            else:
                #doc_list =
list(set(doc_list).intersection(set(term_doc_list))) #list of terms' docIDs
combined by a logical AND---intersection done on the fly
                doc_list = list(set(doc_list).union(set(term_doc_list)))
                #print len(doc_list)
            '''
        doc_list = [str(doc_id) for doc_id in list(set(doc_list))]
        return doc_list    #list of docID containing all terms in a query
```

```
are returned to calling program---since sets support ordered list, when
needed they help in retrieving ranked results

    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

        terms = self.preprocessing()
        scores = defaultdict(list)   #stores scores for each of the
documents----follow algorithm---docID is the key

        inv_index = self.index   #loads the index_file which is in json
form
        qterm_weights = list()   #this is for one query term
        docID = None
        throwawaylist = []
        for term in terms:   #iterating through the query terms
            w_tq = terms.count(term)   #frequency of a term in a query
            qterm_weights.append(w_tq/float(len(terms)))    #normalize term
frequencies
            try:
                  postings_list = inv_index[term] #postings list for a term
in the index---this is a list of {docid:[pos-1,...pos-n]} pairs
            except:
                  continue
            term_docID_list = [int(dict_ID.keys()[0]) for dict_ID in
postings_list]

            for doc in self.cf.docs:
                doc_ID = doc.docID
                if int(doc_ID) in term_docID_list:
                    len_docID = len(util.splitDoc(doc.body))   #total
length of the doc
                    len_positions = 0
                    for term_dict in postings_list:
                        if term_dict.keys()[0] == doc_ID:
                            len_positions = len(term_dict.values())
                            break
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
                    tf = len_positions/float(len_docID)   #division of
query term freq in a docID by total length of the docID
                    tf_idf = tf * self.idf(self.nDocs,len(postings_list))

                    if doc_ID not in list(scores.keys()):
                        scores[doc_ID]=list()
                        scores[doc_ID].append(tf_idf)
                    else:
                        scores[doc_ID].append(tf_idf)
                else:
                    if doc_ID not in list(scores.keys()):
                        scores[doc_ID]=list()
                        scores[doc_ID].append(0.0)
                    else:
                        scores[doc_ID].append(0.0)


        doc_similarity = {}
        for docid, scores_list in scores.iteritems():

doc_similarity[docid]=self.cosine_similarity(qterm_weights,scores_list)

        sorted_scores = sorted(doc_similarity.items(),
key=operator.itemgetter(1), reverse=True)

        top_k = [(i[0],i[1]) for i in sorted_scores[:k]]

        return top_k

    def idf(self, nDocs, nDocs_term):
        ''' compute the inverted document frequency for a given term'''
        #ToDo: return the IDF of the term
        try:
            idf_score = math.log(nDocs, nDocs_term)
        except:
            idf_score = 1.0
        return idf_score

    def dot_product(self, v1, v2):
        return sum(map(lambda x: x[0] * x[1], izip(v1, v2)))
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
    def cosine_similarity(self, vec1, vec2):
        prod = sum(map(lambda x: x[0] * x[1], izip(vec1, vec2)))
        vec1_len = math.sqrt(self.dot_product(vec1, vec1))
        vec2_len = math.sqrt(self.dot_product(vec2, vec2))
        if vec1_len == 0.0 or vec2_len == 0.0:
            return 0.0
        else:
            return round(prod / ((vec1_len * vec2_len)), 2)

def to_ndcg(qrels, q_text, idx_file, tk=10, n=2):
    column_names =['qid', 'docid', 'bool_rel', 'vec_rel']   #for creating a
dataframe for easier data manupilation
    #df_qrels = pd.read_csv('../CranfieldDataset/qrels.text',
names=column_names, sep=' ')   #can test by hard-coding
    df_qrels = pd.read_csv('../CranfieldDataset/qrels.sample',
names=column_names, sep=' ')   #can test by hard-coding
    #df_qrels = pd.read_csv(qrels, names=column_names, sep=' ')
    #print df_qrels


    unique_qids = list(set(list(df_qrels.qid.values)))
    random.shuffle(unique_qids)
    random_qids = unique_qids[0:n]

    qrys =  cranqry.loadCranQry('../CranfieldDataset/query.text')  #qrys is
a dict---for hard-coded testing
    #qrys =  cranqry.loadCranQry(q_text)  #qrys is a dict

    qrys_ids = [key for key,val in qrys.iteritems()]

    II = index.InvertedIndex()
    index_file = II.load("index_file.json")     #for hard-coded testing
    #index_file = II.load(idx_file)


    vec_agg_ndcg, bool_agg_ndcg = list(), list()   #for storing aggregate
ndcg scores
    for qid in random_qids:
      print qid
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
        df_qid = df_qrels[df_qrels["qid"] == qid]   #dataframe for one
query id---comparison of an integer qid in a string qid

      qid_docids = list(df_qid['docid'])    #list of doc ids for a randomly
chosen query id from qrels.text---to be used for ndcg_score
      print qid_docids

        st_qid = str(qid)   #very important----the decimal number in
random_qids should be matched the octal numbers in the cranfield dataset

        if len(st_qid) == 1:    #for handing decimal to octal qid
conversion
            st_qid = "00" + st_qid
        elif len(st_qid) == 2:
            st_qid = "0" + st_qid
        else:
            st_qid = st_qid

        if st_qid in qrys_ids:
            qp = QueryProcessor(qrys[st_qid].text, index_file, 'cran.all')

            bool_array = qp.booleanQuery()
            vec_array = qp.vectorQuery(10)    #change back to 'tk'
            print bool_array
            bool_array = [int(v) for v in bool_array]
            print bool_array
            #ndcg for boolean model
            bool_list = [(0,0)] * 10   #change back to tk

            idx = 0
            for doc_id in bool_array:
                  if doc_id in qid_docids:   #iteratively check if a docid
returned by the vector model is present in qrels.text for the specific
query(qid)
                        #y_true[idx] = 1
                        bool_list[idx] = (1,1)
                        idx += 1
                  else:
                        bool_list[idx] = (0,1)
                  if idx == 10:
```

```
                        break
            #print bool_list

            y_true = [int(bool_id[0]) for bool_id in bool_list]
            y_score = [int(bool_id[1]) for bool_id in bool_list]
            print "bool", y_true
            print "bool", y_score

            bool_agg_ndcg.append(metrics.ndcg_score(y_true, y_score, 10))

            #ndcg for vector model
            print vec_array
            y_score = [vec_id[1] for vec_id in vec_array]   #y_score--to be
passed to ndcg_score is the list of cosine similarity scores
            vec_ids = [int(vec_id[0]) for vec_id in vec_array]   #list of
docids from the list of tuples of the form (docid, similarity_score)
            #print vec_ids
            y_true = [0] * 10   ##added on 0317---change back to tk
            idx = 0
            for doc_id in vec_ids:
                  if doc_id in qid_docids:   #iteratively check if a docid
returned by the vector model is present in qrels.text for the specific
query(qid)
                        y_true[idx] = 1
                        idx += 1
            print "vec", y_true
            print "vec", y_score
            vec_agg_ndcg.append(metrics.ndcg_score(y_true, y_score, 10))

            del qp     ##garbage collection

    return bool_agg_ndcg, vec_agg_ndcg


def test():
    ''' test your code thoroughly. put the testing cases here'''
    #test cases are included in the 'query' method--a sanity check we did
is docIDs returned by the vector model are a subset of the ones from
boolean model
    print 'Pass'
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
def query():
    ''' the main query processing program, using QueryProcessor'''

    II = index.InvertedIndex()
    index_file = sys.argv[1]
    index_file = II.load(index_file)

    proc_alg = sys.argv[2]
    proc_alg = proc_alg

    q_text = sys.argv[3]
    q_text = q_text

    qid = sys.argv[4]
    qid = qid

    qrys =  cranqry.loadCranQry(q_text)  #qrys is a dict
    #qrys =  cranqry.loadCranQry('../CranfieldDataset/query.text')  #can
also be hard-coded like this one

    #qid = '069'   #example of hard-coding a query id
    qp = QueryProcessor(qrys[qid].text, index_file, 'cran.all')   #qid, and
index_file are to be passed by the user

    #print qp.booleanQuery()
    if proc_alg == '0':
        qp.booleanQuery()
        print qp.booleanQuery()
    elif proc_alg == '1':
        qp.vectorQuery(3)    #returning top 3 ranked results for the vector
model
        print qp.vectorQuery(3)


    # 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


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

Batch_eval.py

```
'''
Authors: Manas Gaur, Amanuel Alambo
Instructor: Dr. Keke Chen

a program for evaluating the quality of search algorithms using the vector
```

Manas Gaur, Amanuel Alambo
Information Retrieval, CS 7800
Project-1
A Simple Search Engine

```
model

it runs over all queries in query.text and get the top 10 results,
and then qrels.text is used to compute the NDCG metric

usage:
    python batch_eval.py index_file query.text qrels.text n

    output is the average NDCG over all the queries for boolean model and
vector model respectively.
      also compute the p-value of the two ranking results.
'''
import metrics
import query
from scipy.stats import ttest_ind
import sys
import math

def eval():
      # ToDo
      idx_file = sys.argv[1]   #index file
      idx_file = idx_file

      q_text = sys.argv[2]   #query text
      q_text = q_text

      qrels = sys.argv[3]   #qrels file
      qrels = qrels

      n = sys.argv[4]   #qrels file
      n = int(n)    #typecasting into int, so it can be processed by
downstream tasks with no complaint

      #n=2  #can test by hard-coding
      #bool_ndcg_scores = list()
      #vec_ndcg_scores = list()
      for i in range(1):
            bool_ndcg_score, vec_ndcg_score = query.to_ndcg(qrels, q_text,
idx_file, 10, n)
```

```
            #ndcg_scores.append(avg_ndcg_score)
            bool_ndcg_scores = [0.99 if math.isnan(e) else e for e in
bool_ndcg_score]
            vec_ndcg_scores = [0.99 if math.isnan(e) else e for e in
vec_ndcg_score]
            print bool_ndcg_scores
            print vec_ndcg_scores
            print ttest_ind(bool_ndcg_scores, vec_ndcg_scores)   #checking
wilcoxon for the first two lists of ndcg_scores

      print 'Done'

if __name__ == '__main__':
    eval()
```