Adam Horvath-Smith

Code Dump

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

   output is the average NDCG over all the queries

'''

from cranqry import loadCranQry
from random import choice
from query import QueryProcessor
from cran import CranFile
from index import InvertedIndex, IndexItem, Posting
from metrics import ndcg_score
from scipy.stats import wilcoxon, ttest_ind
from sys import argv

# Values to set (using the init function)
n = 10
index_file = ""
query_path = ""
```

```
qrels_path = ""

def eval():

    # Algorithm:
        # Pick N random samples from query.txt
        # Get top 10 results from bool query for each rnd query
        # Get top 10 results from vector query for each rnd query
        # Compute NDCG btn bool query results and qrels.txt
        # Compute NDCG btn vector query results and qrels.txt
        # Get p-value btn bool and vector

    # Get the query collection
    qc = loadCranQry(query_path)
    poss_queries = list(qc)

    # Load up the inverted index
    ii = InvertedIndex()
    ii.load(index_file)

    # Load up the document collection
    cf = CranFile("cran.all")

    # Get ground-truth results from qrels.txt
    with open(qrels_path) as f:
        qrels = f.readlines()

    # Index qrels into a dict
    qrel_dict = {}
```

```
for qrel in qrels:
    qrel_split = qrel.split()
    if int(qrel_split[0]) in qrel_dict:
        qrel_dict[int(qrel_split[0])].append(int(qrel_split[1]))
    else:
        qrel_dict[int(qrel_split[0])] = [int(qrel_split[1])]

# Run over N random queries, collecting NDCGs
bool_ndcgs = []
vector_ndcgs = []
for _ in range(n):
    # Get random query ID
    query_id = choice(poss_queries)

    # Get the query
    if 0 < int(query_id) < 10:
        query_id = '00' + str(int(query_id))
    elif 9 < int(query_id) < 100:
        query_id = '0' + str(int(query_id))
    try:
        query = qc[query_id].text
    except KeyError:
        print("Invalid query id", query_id)
        return

    # Initialize the query processor
    qp = QueryProcessor(query, ii, cf)

    # Run bool query
```

```
    bool_result = qp.booleanQuery()[:10]

    # Run vector query
    vector_result = qp.vectorQuery(10)

    # Pull top 10 ground-truth results from qrels dict
    gt_results = qrel_dict[poss_queries.index(query_id)+1][:10]

    # Compute NDCG for bool query
    # NOTE: There is no weighting on the bool query, so give all an even 1
    truth_vector = list(map(lambda x: x in gt_results, bool_result))
    bool_ndcg = ndcg_score(truth_vector, [1] * len(truth_vector), k=len(truth_vector))

    # Compute NDCG for vector query
    vector_docs = []
    vector_scores = []
    for v in vector_result:
        vector_docs.append(v[0])
        vector_scores.append(v[1])
    truth_vector = list(map(lambda x: x in gt_results, vector_docs))
    vector_ndcg = ndcg_score(truth_vector, vector_scores, k=len(truth_vector))

    # Accumulate NDCGs
    bool_ndcgs.append(bool_ndcg)
    vector_ndcgs.append(vector_ndcg)

# Average out score lists
bool_avg = 0
for bool in bool_ndcgs:
```

```
        bool_avg += bool
    bool_avg /= len(bool_ndcgs)

    vector_avg = 0
    for vector in vector_ndcgs:
        vector_avg += vector
    vector_avg /= len(vector_ndcgs)

    # Present averages and p-values
    print("Boolean NDCG average:", bool_avg)
    print("Vector NDCG average:", vector_avg)
    if n > 19:
        print("Wilcoxon p-value:", wilcoxon(bool_ndcgs, vector_ndcgs).pvalue)
    else:
        print("Wilcoxon p-value: Sample size too small to be significant")
    print("T-Test p-value:", ttest_ind(bool_ndcgs, vector_ndcgs).pvalue)

def init():
    global n
    global index_file
    global query_path
    global qrels_path

    # Ensure args are valid
    if len(argv) is not 5:
        print("Syntax: python batch_eval.py <index-file-loc> <query-loc> <qrels-loc> <n>")
        return False

    # Grab arguments
```

```
    index_file = argv[1]
    query_path = argv[2]
    qrels_path = argv[3]
    n = int(argv[4])

    # Ensure we have enough test cases
    if n < 2:
        print("N value is too small. Try again.")
        return False

    return True

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

'''

processing the special format used by the Cranfield Dataset



'''
from doc import Document


class CranFile:
    def __init__(self, filename):
        self.docs = []
```

```
cf = open(filename)
docid = ''
title = ''
author = ''
body = ''

for line in cf:
    if '.I' in line:
        if docid != '':
            body = buf
            self.docs.append(Document(docid, title, author, body))
        # start a new document
        docid = line.strip().split()[1]
        buf = ''
    elif '.T' in line:
        None
    elif '.A' in line:
        title = buf # got title
        buf = ''
    elif '.B' in line:
        author = buf # got author
        buf = ''
    elif '.W' in line:
        buf = '' # skip affiliation
    else:
        buf += line
self.docs.append(Document(docid, title, author, buf)) # the last one
```

```
if __name__ == '__main__':
    ''' testing '''

    cf = CranFile (r"..\CranfieldDataset\cran.all")
    for doc in cf.docs:
        print (doc.docID, doc.title)
    print (len(cf.docs))



'''
  handling the specific input format of the query.text for the Cranfield data
'''



class CranQry:
    def __init__(self, qid, text):
        self.qid = qid
        self.text = text

def loadCranQry(qfile):
    queries = {}
    f = open(qfile)
    text = ''
    qid = ''
    for line in f:
        if '.I' in line:
            if qid !='':
                queries[qid] = CranQry(qid, text)
                #print 'qid:', qid, text
```

```
            qid = line.strip().split()[1]
            text = ''
        elif '.W' in line:
            None
        else:
            text += line
    queries[qid] = CranQry(qid, text)
    return queries

def test():
    '''testing'''
    qrys =  loadCranQry(r'..\CranfieldDataset\query.text')
    for q in qrys:
        print(q, qrys[q].text)
    print(len(qrys))

if __name__ == '__main__':
    test()

'''

The document class, containing information from the raw document and possibly other tasks

The collection class holds a set of docuemnts, indexed by docID

'''

class Document:
    def __init__(self, docid, title, author, body):
```

```
        self.docID = docid
        self.title = title
        self.author = author
        self.body = body


    # add more methods if needed


class Collection:
    ''' a collection of documents'''

    def __init__(self):
        self.docs = {} # documents are indexed by docID

    def find(self, docID):
        ''' return a document object'''
        if self.docs.has_key(docID):
            return self.docs[docID]
        else:
            return None

    # more methods if needed


'''
by Mathieu Blondel
'''
```

```
import numpy as np


def dcg_score(y_true, y_score, k=10, gains="exponential"):
    """Discounted cumulative gain (DCG) at rank k

    Parameters

    ----------

    y_true : array-like, shape = [n_samples]

        Ground truth (true relevance labels).

    y_score : array-like, shape = [n_samples]

        Predicted scores.

    k : int

        Rank.

    gains : str

        Whether gains should be "exponential" (default) or "linear".

    Returns

    -------

    DCG @k : float

    """

    order = np.argsort(y_score)[::-1]

    y_true = np.take(y_true, order[:k])


    if gains == "exponential":

        gains = 2 ** y_true - 1

    elif gains == "linear":

        gains = y_true

    else:

        raise ValueError("Invalid gains option.")


    # highest rank is 1 so +2 instead of +1
```

```
    discounts = np.log2(np.arange(len(y_true)) + 2)
    return np.sum(gains / discounts)


def ndcg_score(y_true, y_score, k=10, gains="exponential"):
    """Normalized discounted cumulative gain (NDCG) at rank k
    Parameters
    ----------
    y_true : array-like, shape = [n_samples]
        Ground truth (true relevance labels).
    y_score : array-like, shape = [n_samples]
        Predicted scores.
    k : int
        Rank.
    gains : str
        Whether gains should be "exponential" (default) or "linear".
    Returns
    -------
    NDCG @k : float
    """
    best = dcg_score(y_true, y_true, k, gains)
    actual = dcg_score(y_true, y_score, k, gains)
    return actual / best


'''

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 cran import CranFile
from pickle import dump, load
from math import log10, sqrt
from sys import argv


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()
```

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

    def term_freq(self):
        ''' return the term frequency in the document'''

        # The number of times a term is in a document corresponds to
        #   the length of the position list
        return len(self.positions)

    # For testing purposes...
    def __repr__(self):
        return str(self.positions)



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 docid in self.posting:
            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

        # We already have the postings in posting dict. Store the sorted docID keys
        #   in sorted_postings list, for easier reference in postings dict.
        self.sorted_postings = sorted(self.posting)

        # We sort the positions of each posting in place
        for doc in self.posting:
            self.posting[doc].sort()



class InvertedIndex:

    def __init__(self):
        self.items = {} # list of IndexItems
        self.doc_tfidf = {} # tf-idf of every term in every doc
        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'''

        # Using the SPIMI algorithm as defined at
        # https://nlp.stanford.edu/IR-book/html/htmledition/single-pass-in-memory-indexing-1.html

        # Each term in a doc has its own index item!!!

        # Preprocess first...
```

```
        # Call tokenize_doc to convert doc title and body into tokenized list, in lowercase
        # Do remove stopwords and stemming as expected

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

        # Then go term-by-term and create the index. Use algorithm to track which terms already in index, add new ones if not. If we create a new index item, add it to the self.items dict!!!

        # ---

        # Increment number of documents indexed
        self.nDocs += 1

        # Grab title and body of doc, merge into one string
        doc_string = doc.body

        # Tokenize and lowercase doc into list form
        token_list = util.tokenize_doc(doc_string)

        # Helper function to replace stopwords with empty string
        def remove_stop_word(tok):
            return "" if util.isStopWord(tok) else tok

        # Remove the stopwords from both positional list and token list
        token_list_no_stopword = list(map(remove_stop_word, token_list))
```

```
    # Stem the words
    stemmed_token_list = list(map(lambda tok: util.stemming(tok), token_list_no_stopword))

    # Note that the stemmed tokens are now our terms
    for pos, term in enumerate(stemmed_token_list):
        # Skip over stopwords, now replaced by ""
        if term == "": continue

        # If this term has already appeared, update the existing posting
        if not term in self.items:
            self.items[term] = IndexItem(term)
        self.items[term].add(int(doc.docID), pos)



def sort(self):
    ''' sort all posting lists by docID'''
    #ToDo

    # The actual sort is implemented in IndexItem. Just call it here.
    for item in self.items:
        self.items[item].sort()

def find(self, term):
    return self.items[term] if term in self.items else None

def save(self, filename):
    ''' save to disk'''
    # ToDo: using your preferred method to serialize/deserialize the index
```

```
    # Combine items dict and nDocs into a list so they can be pickled together
    to_pickle = [self.items, self.nDocs, self.doc_tfidf]

    # Use Pickle to dump the index to a file
    with open(filename, 'wb') as out:
        dump(to_pickle, out)

def load(self, filename):
    ''' load from disk'''
    # ToDo

    # Load data back from pickled file
    with open(filename, 'rb') as inf:
        file_read = load(inf)
        self.items = file_read[0]
        self.nDocs = file_read[1]
        self.doc_tfidf = file_read[2]

def idf(self, term):
    ''' compute the inverted document frequency for a given term'''
    #ToDo: return the IDF of the term

    # IDF of term t is log(total # of docs / # docs with t in it)
    return log10(self.nDocs / len(self.items[term].posting)) \
        if term in self.items else 0

def compute_tfidf(self):
    """ pre-compute tf-idf vectors for each word in each doc """
    # Handle empty items with empty fector
```

```
self.doc_tfidf[471] = {}
self.doc_tfidf[995] = {}

# Compute tf-idf vector for every other doc
for iter in range(self.nDocs):
    doc = iter + 1
    word_vector = {}

    # Ignore docs we know to be empty
    if doc in (471, 995): continue

    for word in self.items:
        # Get tf
        try:
            tf = self.find(word).posting[doc].term_freq()
        except KeyError: # if not in doc
            tf = 0

        # Get idf
        idf = self.idf(word)

        # Calculate tf-idf; add to current dict
        word_vector[word] = log10(1 + tf) * idf

    # Normalize the word vector
    accum = 0
    for word in word_vector:
        accum += word_vector[word]**2
    accum = sqrt(accum)
```

```
        for word in word_vector:
            word_vector[word] /= accum

        self.doc_tfidf[doc] = word_vector


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

    ####### TEST CASES FOR INVERTED INDEX CLASS #######

    # Get all documents from cran.all--let Cranfile object handle this
    cf = CranFile("cran.all")

    # Build an inverted index object
    ii = InvertedIndex()

    # Index one document
    ii.indexDoc(cf.docs[0])

    # The first temr should be "experiment" (verified by printing contents of II)
    #   We want to ensure that find() finds it
    index_item = ii.find("experiment")
    print("Result of find:", index_item.term, index_item.posting)

    # Next, sort to ensure that it works
    # TODO: figure out what this should doc
    ii.sort()
```

```
print("Sorted!")

# Get the IDF of the term "experiment"
#   Following the formula from our slides, this should be 0
print("IDF:", ii.idf("experiment"))

# Add back in the rest of Cranfield dataset
for doc in cf.docs[1:]:
    ii.indexDoc(doc)

# Re-do find now that we have more things in the index
index_item = ii.find("experiment")
print("Result of find:", index_item.term, index_item.posting)

# Ensure sort works on larger index
# Next, sort to ensure that it works
# TODO: figure out what this should doc
ii.sort()
print("Sorted!")

# Calculate IDF with larger index
# Get the IDF of the term "experiment"
#   Following the formula from our slides, this should be 0
print("IDF:", ii.idf("experiment"))

# Get the tfidf dict
ii.compute_tfidf()

# Save off our index
```

```
ii.save("index.pkl")

# Read back in the index, ensure they are the same
ii_from_file = InvertedIndex()
ii_from_file.load("index.pkl")

# Cannot determine if the actual items are equal objects,
#   so just ensure the stats are the same
# print("Load matches saved items:", ii.items == ii_from_file.items)
print("Load matches saved number of docs:", ii.nDocs == ii_from_file.nDocs)
print("Load matches saved IDF for 'experiment':",
   ii.idf("experiment") == ii_from_file.idf("experiment"))
print("Load matches saved find term for 'experiment':",
   ii.find("experiment").term == ii_from_file.find("experiment").term)
print("Load matches saved find posting for 'experiment':",
   str(ii.find("experiment").posting) == str(ii_from_file.find("experiment").posting))

####### TEST CASES FOR POSTING CLASS #######

# Create test posting
p = Posting(docID=1)

# Test adding a position
p.append(3)
print("Position appended to posting:", p.positions == [3])

# Add position out of order, ensure sort works
p.append(1)
print("Append is initially out-of-order:", p.positions == [3, 1])
```

```
p.sort()
print("Sort correctly sorts postings:", p.positions == [1, 3])

# Ensure we can merge in new postings
to_merge = [4, 5, 6]
p.merge(to_merge)
print("Merge correctly merges:", p.positions == [1, 3, 4, 5, 6])

# Ensure term frequency is correctly
print("Term frequency correctly counts postings:", p.term_freq() == 5)

####### TEST CASES FOR INDEX ITEM CLASS #######

# Create index item
iitem = IndexItem("abc")

# Add value to index item
iitem.add(0, 40)
print("Document added to item:", 0 in iitem.posting)
print("Posting created for document in item:",
   type(iitem.posting[0]) == type(Posting(5)))

####### ADDITIONAL TEST CASES #######

print("\nTHE FOLLOWING ARE BASED ON THE GIVEN TEST QUESTIONS")

# Act on the assumption all words are stemmed
#   This should be done in the tokenize part of util
#   The idea was to re-stem all words and ensure they equal the words
```

```
    #     in the index, but some double-stemmings differ anyway.

    # Ensure stopwords were removed
    from nltk.stem.porter import PorterStemmer
    with open("stopwords") as f:
        stopwords = f.readlines()
    s = PorterStemmer()
    stopword_vector = list(
        map(lambda x: s.stem(x.strip()) in ii.items.items(), stopwords))
    print("All stopwords removed from index:", not any(stopword_vector))

    # Print number of terms in dict--Dr. Chen can ensure this is right
    print("Number of terms in dictionary:", len(ii.items))

    # Print average size of postings--Dr. Chen can ensure this makes sense
    sum = 0
    posting_count = 0
    for item in ii.items.values():
        for posting in item.posting.values():
            sum += len(posting.positions)
            posting_count += 1
    print("Average posting length:", sum/posting_count)


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
```

```
    # Ensure args are valid
    if len(argv) != 3:
        print("Syntax: python index.py <cran.all path> <index-save-location>")
        return

    # Grab arguments
    file_to_index = argv[1]
    save_location = argv[2]

    # Index file
    print("Indexing documents from", file_to_index + "...")
    cf = CranFile(file_to_index)
    ii = InvertedIndex()
    for doc in cf.docs:
        ii.indexDoc(doc)

    # Sort index before saving
    ii.sort()

    # Compute tf-idf vector representations for each doc
    ii.compute_tfidf()

    # Save off index
    ii.save(save_location)
    print("Index saved to", save_location + "!")


if __name__ == '__main__':
    #test()  # Uncomment to run tests
```

```
    indexingCranfield()



'''
by Mathieu Blondel
'''

import numpy as np

def dcg_score(y_true, y_score, k=10, gains="exponential"):
    """Discounted cumulative gain (DCG) at rank k
    Parameters
    ----------
    y_true : array-like, shape = [n_samples]
        Ground truth (true relevance labels).
    y_score : array-like, shape = [n_samples]
        Predicted scores.
    k : int
        Rank.
    gains : str
        Whether gains should be "exponential" (default) or "linear".
    Returns
    -------
    DCG @k : float
    """
    order = np.argsort(y_score)[::-1]
    y_true = np.take(y_true, order[:k])

    if gains == "exponential":
```

```
        gains = 2 ** y_true - 1
    elif gains == "linear":
        gains = y_true
    else:
        raise ValueError("Invalid gains option.")

    # highest rank is 1 so +2 instead of +1
    discounts = np.log2(np.arange(len(y_true)) + 2)
    return np.sum(gains / discounts)


def ndcg_score(y_true, y_score, k=10, gains="exponential"):
    """Normalized discounted cumulative gain (NDCG) at rank k
    Parameters
    ----------
    y_true : array-like, shape = [n_samples]
        Ground truth (true relevance labels).
    y_score : array-like, shape = [n_samples]
        Predicted scores.
    k : int
        Rank.
    gains : str
        Whether gains should be "exponential" (default) or "linear".
    Returns
    -------
    NDCG @k : float
    """
    best = dcg_score(y_true, y_true, k, gains)
    actual = dcg_score(y_true, y_score, k, gains)
```

```
    return actual / best if best != 0 else 0



'''
Peter Norvig's python implementation of Spelling Corrector

'''




import re
from collections import Counter

def words(text): return re.findall(r'\w+', text.lower())

WORDS = Counter(words(open('big.txt').read()))

def P(word, N=sum(WORDS.values())):
    "Probability of `word`."
    return WORDS[word] / N

def correction(word):
    "Most probable spelling correction for word."
    return max(candidates(word), key=P)

def candidates(word):
    "Generate possible spelling corrections for word."
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])
```

```
def known(words):
    "The subset of `words` that appear in the dictionary of WORDS."
    return set(w for w in words if w in WORDS)

def edits1(word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    "All edits that are two edits away from `word`."
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))


'''
query processing

'''

from norvig_spell import correction
from index import InvertedIndex, IndexItem, Posting
from cran import CranFile
from cranqry import loadCranQry
from math import log10, sqrt
```

```
from collections import Counter
from sys import argv
import util

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 = collection

    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

        # Tokenize and lowercase doc into list form
        token_list = util.tokenize_doc(self.raw_query)

        # Helper function to replace stopwords with empty string
        def remove_stop_word(tok):
            return "" if util.isStopWord(tok) else tok

        # Correct spelling of each word
        tokens_corrected_spell = list(map(lambda tok: correction(tok),
```

```
        token_list))

    # Remove the stopwords from both positional list and token list
    token_list_no_stopword = list(map(remove_stop_word,
        tokens_corrected_spell))

    # Stem the words
    stemmed_token_list = list(map(lambda tok: util.stemming(tok),token_list_no_stopword))

    return stemmed_token_list



  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

    # Ref: https://nlp.stanford.edu/IR-book/html/htmledition/processing-boolean-queries-1.html

    # Parse position of OR and NOT
    #   We must do this before stopwords removed
    or_list = list(map(lambda t: t.lower() == "or", self.raw_query.split()))
    or_positions = []
    for idx, bool in enumerate(or_list):
        if bool: or_positions.append(idx)

    not_list = list(map(lambda t: t.lower() == "not", self.raw_query.split()))
    not_positions = []
    for idx, bool in enumerate(not_list):
```

```
    if bool: not_positions.append(idx)

# Parse out parenthesis positions
open_paren_list = list(map(lambda t: '(' in t, self.raw_query.split()))
close_paren_list = list(map(lambda t: ')' in t, self.raw_query.split()))
if len(open_paren_list) is not len(close_paren_list):
    print("Error: Parenthesis mismatch.")
    return
open_paren_positions = []
close_paren_positions = []
for idx, bool in enumerate(open_paren_list):
    if bool: open_paren_positions.append(idx)
for idx, bool in enumerate(close_paren_list):
    if bool: close_paren_positions.append(idx)

# Get preprocessed query
clean_query = self.preprocessing()

# Handle precedence of parens
for idx, pos in enumerate(open_paren_positions):
    # Get the actual subquery within parens
    subquery = clean_query[open_paren_positions[idx] : close_paren_positions[idx]+1]

    # Run bool query on subquery
    subquery_result = self.bool_query_helper(subquery, not_positions, or_positions)

    # Replace subquery with completed postings
    clean_query[open_paren_positions[idx]] = subquery_result
```

```
        # Replace rest of subquery with empty strings (will be skipped)
        for i, w in enumerate(clean_query[open_paren_positions[idx]+1 : close_paren_positions[idx]+1]):
            clean_query[i+open_paren_positions[idx]+1] = ''

    # Process the remainder of the query
    answer = self.bool_query_helper(clean_query, not_positions, or_positions)

    return answer



def bool_query_helper(self, query, not_positions, or_positions):
    # Mark whether this is the first word in the query
    first_word = True

    # Get posting for first term to start the list
    master_postings = []

    # Get postings for rest of query terms
    for idx, word in enumerate(query):
        # Skip any empty stopword positions
        if word == '': continue

        # Merge in any existing postings
        if type(word) is type([]):
            # If a not query
            if idx-1 in not_positions:
                word = [n for n in list(range(1, self.index.nDocs+1))
                    if n not in word]
```

```
    # If master_postings empty or this is an or query
    if first_word or idx-1 in or_positions:
        master_postings.extend(word)
        master_postings = sorted(list(set(master_postings)))
        first_word = False
        continue

    # Otherwise, just merge the posting lists
    master_postings = [posting for posting in word
        if posting in master_postings]

    continue

# Get the containing index item
index_item = self.index.find(word)

# Get docs where the word is posted
if index_item:
    current_postings = index_item.sorted_postings[:]
else: current_postings = []

# If an or query, just append current to master
if idx-1 in or_positions:
    master_postings.extend(current_postings)
    master_postings = sorted(list(set(master_postings)))
    continue

# Negate if last position is a not
if idx-1 in not_positions:
```

```
            current_postings = [n for n in list(range(1, self.index.nDocs+1))
                if n not in current_postings]

        # Handle case where this is the first thing in the list
        if first_word:
            master_postings = current_postings
            first_word = False
            continue

        # Merge the current postings into master postings
        master_postings = [posting for posting in current_postings
            if posting in master_postings]

    return master_postings



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

    # May need to let x% of words match here to get any matches
    #   (do same for bool)

    # For each term in query...
        # Grab the tf-idf for each word in each doc
        # Hold on to all doc ids that contain (most?) words in query
        # Compute cosine sim and rank results
```

```
# Get preprocessed query
clean_query = self.preprocessing()

# Get IndexItems for each term in the query
#   Hold on to these in a list so we make sure each term in doc

# Hold on to the number of times each word appears in a doc
doc_dict = {}
for word in clean_query:
   # Skip any empty stopword positions
   if word == '': continue

   # Add in the docs
   word_lookup = self.index.find(word)
   if word_lookup is None: continue

   for doc in word_lookup.posting:
      if doc not in doc_dict:
         doc_dict[doc] = 1
      else: doc_dict[doc] += 1

# Compute the vector representation for each doc, using tf-idf for
#   EVERY possible word
# --> This is pre-computed in the inverted index
tfidf_dict = self.index.doc_tfidf

# Compute the cosine score between each doc and the query
# Ref: https://nlp.stanford.edu/IR-book/html/htmledition/computing-vector-scores-1.html
```

```
scores = {}
word_count_query = Counter(clean_query)
for word in clean_query:
    # Skip any empty stopword positions
    if word == '': continue

    # Get word tf-idf
    tf = word_count_query[word]
    idf = self.index.idf(word)
    tfidf = log10(1+tf) * idf

    # Count up the scores for doc weights
    word_lookup = self.index.find(word)
    if word_lookup is None: continue

    for doc in doc_dict:
        # Get the posting for the word
        cur_posting = word_lookup.posting

        # Calculate the score
        if doc not in cur_posting:
            score = 0
        else:
            score = tfidf * cur_posting[doc].term_freq()

        # Add up the scores
        if doc in scores:
            scores[doc] += score
        else:
```

```
                scores[doc] = score

        # Normalize scores by doc length
        for doc in scores:
            scores[doc] /= len(self.docs.docs[doc-1].body.split())

        # Sort the scores by score
        sorted_scores = sorted(scores.items(), reverse=True, key=lambda x: x[1])

        # Return top k scores
        return sorted_scores[:k]


def test(index_loc, cran_loc, qrels_loc):
    ''' test your code thoroughly. put the testing cases here'''

    ##### SETUP ITEMS #####

    # Grab index file to restore II
    ii = InvertedIndex()
    ii.load(index_loc)

    # Get the document collection
    cf = CranFile(cran_loc)

    # Get ground-truth results from qrels.txt
    with open(qrels_loc) as f:
        qrels = f.readlines()
```

```
# Index qrels into a dict
qrel_dict = {}
for qrel in qrels:
    qrel_split = qrel.split()
    if int(qrel_split[0]) in qrel_dict:
        qrel_dict[int(qrel_split[0])].append(int(qrel_split[1]))
    else:
        qrel_dict[int(qrel_split[0])] = [int(qrel_split[1])]

##### INITIAL TEST ITEMS #####
print("TESTS BASED ON SUGGESTED TESTING POINTS")

# Ensure tf is correct
#   Find a random word and check TF value against what is manually done
posting_list = ii.find("experiment").posting
tf_vector = []
for posting in posting_list:
    tf_vector.append(len(posting_list[posting].positions) \
        == posting_list[posting].term_freq())
print("TF is computed correctly:", all(tf_vector))

# Ensure idf is correct
print("IDF is computed correctly:", log10(ii.nDocs / len(posting_list)) \
    == ii.idf("experiment"))

# As both tf and idf are correct, and tf-idf is a product of the two,
#   it is reasonable to assume tf-idf is computed correctly

##### BOOL QUERY TESTS #####
```

```
    # Here, I use very specific boolean queries to ensure that a
    #   limited number of documents are returned
    print("\nBOOL QUERY TESTS")

    # Ensure that the exact title of doc 8 matches for doc 8
    doc8 = "measurements of the effect of two-dimensional and three-dimensional roughness elements
on boundary layer transition"
    qp1 = QueryProcessor(doc8, ii, cf)
    print("Bool query matches on exact title:", qp1.booleanQuery() == [8])

    # Ensure that bool query matches very specific AND query
    qp2 = QueryProcessor("hugoniot and infinitesimally", ii, cf)
    print("Bool query matches on specific AND query ('hugoniot and infinitesimally'):",
qp2.booleanQuery() == [329])

    # Test that an OR query is handled properly
    #   Both gravel and stagnation have completely distinct postings lists.
    #   OR should merge them.
    gravel_postings = ii.find("gravel").sorted_postings[:]
    stag_postings = ii.find("stagnat").sorted_postings[:]
    gravel_postings.extend(stag_postings)
    qp3 = QueryProcessor("gravel or stagnation", ii, cf)
    print("Bool query successfully handles OR ('gravel or stagnation'):",
        qp3.booleanQuery() == sorted(gravel_postings))

    # Test that NOT is handled properly
    #   The posting list for "diameter" is a subset of "slipstream" postings
    #   (oddly enough). To test this works, do "slipstream and not diameter"
```

```
#   and we chould get slipstream's postings minus those of diameter.
slip_postings = ii.find("slipstream").sorted_postings[:]
diam_postings = ii.find("diamet").sorted_postings[:]
slip_not_diam = [t for t in slip_postings if t not in diam_postings]
print("Bool query successfully handles NOT ('slipstream and not diameter'):",
   QueryProcessor("slipstream and not diameter", ii, cf).booleanQuery() \
     == slip_not_diam)

# Ensure AND/OR order doesn't matter
print("Bool query can handle query regardless of AND order ('a and b' = 'b and a'):",
   QueryProcessor("slipstream and diameter", ii, cf).booleanQuery() \
     == QueryProcessor("diameter and slipstream", ii, cf).booleanQuery())
print("Bool query can handle query regardless of OR order ('a or b' = 'b or a'):",
   QueryProcessor("slipstream or diameter", ii, cf).booleanQuery() \
     == QueryProcessor("diameter or slipstream", ii, cf).booleanQuery())

# Ensure that the presence of parens does not change query results
print("Bool query can handle query regardless of parens ('slipstream and diameter'):",
   QueryProcessor("slipstream and diameter", ii, cf).booleanQuery() \
     == QueryProcessor("(slipstream and diameter)", ii, cf).booleanQuery())

# Ensure parentheses do not change order of processing for AND-AND and OR-OR queries
print("Bool query AND is accociative ('(a and b) and c' = 'a and (b and c)'):",
   QueryProcessor("(slipstream and diameter) and thrust", ii, cf).booleanQuery() \
     == QueryProcessor("slipstream and (diameter and thrust)", ii, cf).booleanQuery())
print("Bool query OR is accociative ('(a or b) or c' = 'a or (b or c)'):",
   QueryProcessor("(slipstream or diameter) or thrust", ii, cf).booleanQuery() \
     == QueryProcessor("slipstream or (diameter or thrust)", ii, cf).booleanQuery())
```

```
# Ensure parentheses properly group items
#   Tested by doing the query "manually" by adding/orring the correct terms
part_one = QueryProcessor("conduction and cylinder and gas", ii, cf).booleanQuery()
part_two = QueryProcessor("radiation and gas", ii, cf).booleanQuery()
part_one.extend(part_two)
expected_result = QueryProcessor("hugoniot", ii, cf).booleanQuery()
expected_result.extend(part_one)
print("Bool query parens successfully group conflicting operators:",
    QueryProcessor("(conduction and cylinder and gas) or (radiation and gas) or hugoniot", ii,
cf).booleanQuery() \
      == sorted(list(set(expected_result))))

##### VECTOR QUERY TESTS #####

# For this, just ensure that most of the results are in the expected list
print("\nVECTOR QUERY TESTS")

# Ensure vector query can match on exact title
print("Vector query matches on exact title:", qp1.vectorQuery(1)[0][0] == 8)

# Try a few example queries from query.text
#   As long as one-fifth of t-10 are in gt_result, call it a pass
# Note that queries with larger answer sets were chosen to
#   ensure there were enough to get to one-fifth of ten
qc = loadCranQry("query.text")
poss_queries = list(qc)

# Query 001
result = QueryProcessor(qc["001"].text, ii, cf).vectorQuery(10)
```

```
gt_result = qrel_dict[poss_queries.index("001")+1]
correct_vector = list(map(lambda x: x in gt_result, [x[0] for x in result]))
print("Vector query is at least one-fifth correct for query 001:", sum(
   correct_vector) > 2)

# Query 128
result = QueryProcessor(qc["128"].text, ii, cf).vectorQuery(10)
gt_result = qrel_dict[poss_queries.index("128")+1]
correct_vector = list(map(lambda x: x in gt_result, [x[0] for x in result]))
print("Vector query is at least one-fifth correct for query 128:", sum(
   correct_vector) > 2)

# Query 226
result = QueryProcessor(qc["226"].text, ii, cf).vectorQuery(10)
gt_result = qrel_dict[poss_queries.index("226")+1]
correct_vector = list(map(lambda x: x in gt_result, [x[0] for x in result]))
print("Vector query is at least one-fifth correct for query 226:", sum(
   correct_vector) > 2)

# Query 196
result = QueryProcessor(qc["196"].text, ii, cf).vectorQuery(10)
gt_result = qrel_dict[poss_queries.index("196")+1]
correct_vector = list(map(lambda x: x in gt_result, [x[0] for x in result]))
print("Vector query is at least one-fifth correct for query 196:", sum(
   correct_vector) > 2)

# Query 291
result = QueryProcessor(qc["291"].text, ii, cf).vectorQuery(10)
gt_result = qrel_dict[poss_queries.index("291")+1]
```

```
    correct_vector = list(map(lambda x: x in gt_result, [x[0] for x in result]))
    print("Vector query is at least one-fifth correct for query 291:", sum(
        correct_vector) > 2)


def query():
    ''' 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 vectorQuer
    # 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

    # Ensure args are valid
    if len(argv) is not 5:
        print("Syntax: python query.py <index-file-path> <processing-algorithm> <query.txt path> <query-id>")
        return

    # Grab arguments
    index_file_loc = argv[1]
    processing_algo = argv[2]
    query_file_path = argv[3]
    query_id = argv[4]

    # Grab index file to restore II
    ii = InvertedIndex()
    ii.load(index_file_loc)
```

```
# Get the document collection
cf = CranFile("cran.all")

# Get the query collection
qc = loadCranQry(query_file_path)

# Get the query
if 0 < int(query_id) < 10:
    query_id = '00' + str(int(query_id))
elif 9 < int(query_id) < 100:
    query_id = '0' + str(int(query_id))
try:
    query = qc[query_id].text
except KeyError:
    print("Invalid query id", query_id)
    return

# Initialize a query processor
qp = QueryProcessor(query, ii, cf)

# Do query
if int(processing_algo) is 0:
    print("Results:", ", ".join(str(x) for x in qp.booleanQuery()))
elif int(processing_algo) is 1:
    result = qp.vectorQuery(k=3)
    print("Results:")
    for r in result:
        print("Doc", r[0], "Score", r[1])
else:
```

```
        print("Invalid processing algorithm", processing_algo +
            ". Use 0 (boolean) or 1 (vector).")


if __name__ == '__main__':
    #test("TEST.pkl", "cran.all", "qrels.text")
    query()


'''
   utility functions for processing terms

    shared by both indexing and query processing
'''

from nltk.stem.porter import PorterStemmer
from string import punctuation

### Initialization code that we only want to do once ###
# Read in stop words
stop_words = open('stopwords').read()

# Initialize a stemmer object
stemmer = PorterStemmer()


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

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

def tokenize_doc(doc):
    """ Get each token (split on whitespace); lowercase each token """

    # Split and lower
    tokens = list(map(lambda word: word.lower(), doc.split()))

    # Remove punctuation
    return list(map(lambda word: word.translate(str.maketrans("","",
        punctuation)), tokens))
```