**Index.py**

**import math**

**import pickle**

**import sys**

**import jsonpickle**

**from util import \***

**import doc**

**import cran**

**class Posting:**

**def \_\_init\_\_(self, docID):**

**self.docID = docID**

**self.positions = []**

**self.term\_frequency=0**

**def append(self, pos):**

**self.positions.append(pos)**

**self.term\_frequency=len(pos)**

**def sort(self):**

**''' sort positions'''**

**self.positions.sort()**

**def merge(self, positions):**

**self.positions.extend(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**

**#self.termFrequency=0**

**def add(self, docid, pos):**

**''' add a posting'''**

**if docid not in self.posting.keys():**

**self.posting[docid] = Posting(docid)**

**self.posting[docid].append(pos)**

**class InvertedIndex:**

**def \_\_init\_\_(self):**

**self.items = {} # list of IndexItems**

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

**def Find\_positions(self,docwordlist,word):**

**words\_pos=[]**

**position=0**

**for i in range(len(docwordlist)):**

**if docwordlist[i] == word:**

**words\_pos.append(i+1)**

**return words\_pos**

**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**

**# Consider title and body of Docuemnts for indexing**

**# (1) convert to lower cases,**

**# (2) Tokenizing**

**# (3) remove stopwords,**

**# (4) stemming**

**title=doc.title**

**body=doc.body**

**#First convert both title and body words and lowercase**

**documentWords=title+body**

**lowered\_documentWords=ConvertLowerCase(documentWords)**

**tokenizedWords=Tokenize(lowered\_documentWords)**

**removedstopwords=removeStopWords(tokenizedWords)**

**stemmedWords=stemming(removedstopwords)**

**#print(stemmedWords)**

**self.nDocs=self.nDocs+1**

**documentWord={}**

**# consider the stemmed words for indexing**

**for x in stemmedWords:**

**if(x not in documentWord.keys()):**

**doc\_pos= self.Find\_positions(stemmedWords, x)**

**if(x not in self.items.keys()):**

**indexitemobj=IndexItem(x)**

**indexitemobj.add(self.nDocs,doc\_pos)**

**self.items[x]=indexitemobj**

**else:**

**indexitemobj = IndexItem(x)**

**indexitemobj.add(self.nDocs, doc\_pos)**

**self.items.get(x).add(self.nDocs, doc\_pos)**

**documentWord[x]=doc\_pos**

**#print(documentWord)**

**#self.updateIndex(documentWord)**

**def sort(self):**

**''' sort all posting lists by docID'''**

**#ToDo**

**#sort is being inbuilt based on positions as we use Find\_positions method**

**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**

**print("Number of terms being Saved to Index\_File",self.items.\_\_len\_\_())**

**jsonEncoded = jsonpickle.encode(self)**

**f=open(filename,'w')**

**f.write(jsonEncoded)**

**print("Index File Generated as ",filename)**

**def load(self, filename):**

**''' load from disk'''**

**# ToDo**

**filepointer=open(filename,'r')**

**indexRead=jsonpickle.decode(filepointer.read())**

**return indexRead**

**def idf(self, term):**

**''' compute the inverted document frequency for a given term'''**

**#ToDo: return the IDF of the term**

**idf\_value = self.nDocs/len(self.items[term].posting.keys())**

**self.items[term].idf = math.log(idf\_value,10)**

**# more methods if needed**

**def test(obj):**

**''' test your code thoroughly. put the testing cases here'''**

**#StopWords Removal,lowercase,tokenize,stemming.**

**document="experimental investigation of the aerodynamics of a wing in a slipstream "**

**print("Sample Document considered ::",document)**

**lowered\_documentWords = ConvertLowerCase(document)**

**tokenizedWords = Tokenize(lowered\_documentWords)**

**removedstopwords = removeStopWords(tokenizedWords)**

**stemmedWords = stemming(removedstopwords)**

**stopwords\_Removed=' '.join(removedstopwords)**

**print("Sample Document After Removal of StopWords ::",stopwords\_Removed)**

**print("Terms After stemmed ::",stemmedWords)**

**#loading Index File**

**IndexobjectRead=obj.load("index\_file")**

**print('Loading Index File Test Case Passed')**

**def indexingCranfield(data\_file,indexfile):**

**#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**

**#first load the document file**

**inputdocument= cran.CranFile(data\_file)**

**#create object for invetedIndex class**

**#iterate over the document File and create the index file**

**#calculate idf**

**invertedobj=InvertedIndex()**

**for docs in inputdocument.docs:**

**invertedobj.indexDoc(docs)**

**for x in invertedobj.items:**

**invertedobj.idf(x)**

**invertedobj.save(indexfile)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**#python index.py cran.all index\_file**

**#indexingCranfield('cran.all','index\_file')**

**indexingCranfield(str(sys.argv[1]), str(sys.argv[2]))**

**util.py**

**'''**

**utility functions for processing terms**

**shared by both indexing and query processing**

**'''**

**from nltk import PorterStemmer**

**from nltk.corpus import stopwords**

**from nltk.tokenize import word\_tokenize**

**from norvig\_spell import correction**

**def Tokenize(word):**

**#ntlk function to tokenize the words**

**removable\_chars = [';', ':', '!', '\*','.',',','/']**

**word = ''.join(i for i in word if not i in removable\_chars)**

**return word\_tokenize(word)**

**def spellcheck(word):**

**spellcheckedwords=[]**

**for x in word:**

**wrd=correction(x)**

**spellcheckedwords.append(wrd)**

**return spellcheckedwords**

**def ConvertLowerCase(word):**

**return word.lower()**

**def ReadStopWordsFile():**

**stopwordslist=[]**

**stopwordsFile = open("stopwords", "r")**

**stopwords=stopwordsFile.read()**

**stopwords=stopwords.split('\n')**

**return stopwords**

**def isStopWord(word):**

**''' using the NLTK functions, return true/false'''**

**#check with nltk stopwords as well even after removing from stopword files**

**# ToDo**

**stop\_words = set(stopwords.words("english"))**

**if word in stop\_words:**

**return True**

**else:**

**return False**

**def removeStopWords(words):**

**stopwords=ReadStopWordsFile()**

**stopwordsremoval=[]**

**for x in words:**

**if(x not in stopwords):**

**if(not isStopWord(x)):**

**stopwordsremoval.append(x)**

**return stopwordsremoval**

**def stemming(word):**

**''' return the stem, using a NLTK stemmer. check the project description for installing and using it'''**

**# ToDo**

**porterstem = PorterStemmer()**

**stemmedwords=[]**

**for x in word:**

**stemmedwords.append(porterstem.stem(x))**

**return stemmedwords**

**if \_\_name\_\_ == '\_\_main\_\_':**

**print(ReadStopWordsFile())**

**Query.py**

**import math**

**import random**

**import sys**

**from time import process\_time**

**import cran**

**from cranqry import loadCranQry**

**from index import InvertedIndex**

**from util import \***

**from functools import reduce**

**from heapq import nlargest**

**class QueryProcessor:**

**def \_\_init\_\_(self, query, index, collection,query\_nid):**

**''' index is the inverted index; collection is the document collection'''**

**self.raw\_query = query**

**self.index=index**

**self.docs = collection**

**self.querynumber=query\_nid**

**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**

**querynumber="{0:0=3d}".format(self.querynumber)**

**#print(querynumber)**

**if(querynumber not in self.raw\_query.keys()):**

**print("Query Number does not exists")**

**exit(0)**

**inputquery=self.raw\_query[querynumber].text**

**#print(inputquery)**

**#tokenize the query words**

**lowered\_documentWords=ConvertLowerCase(inputquery)**

**tokenizedWords=Tokenize(lowered\_documentWords)**

**spellcheckedwords=spellcheck(tokenizedWords)**

**removedstopwords=removeStopWords(spellcheckedwords)**

**stemmedWords=stemming(removedstopwords)**

**return stemmedWords**

**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**

**preproces\_terms=self.preprocessing()**

**#print(preproces\_terms)**

**listdocidterms=[]**

**for term in preproces\_terms:**

**if term in self.index.items.keys():**

**documentIDlist=list(self.index.items.get(term).get('posting').keys())**

**listdocidterms.append(documentIDlist)**

**else:**

**listdocidterms.append([])**

**#print(listdocidterms)**

**res = list(reduce(lambda i, j: i & j, (set(x) for x in listdocidterms)))**

**#print(res)**

**return res**

**def vectorQuery(self, k,alt=False):**

**''' 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**

**#if alt false then weight are caluclted as (query)ltc.ltc(document) and alt is true then (query) apc.ltc (docuemnt)**

**preproces\_terms = self.preprocessing()**

**#print(preproces\_terms)**

**#print(preproces\_terms)**

**documents=self.docs**

**#calculate tf-idf of query words**

**#we are gona caculate using tf-idf = termfequency \* idf**

**highest\_term\_fequency=0;**

**if(alt):**

**for x in preproces\_terms:**

**xls=preproces\_terms.count(x)**

**if(xls>highest\_term\_fequency):**

**highest\_term\_fequency=xls**

**qcvector={}**

**for t in preproces\_terms:**

**if(not alt):**

**tf=preproces\_terms.count(t)**

**else:**

**tfk = preproces\_terms.count(t)**

**tf=0.5+((0.5\*tfk)/highest\_term\_fequency)**

**if(t in self.index.items):**

**if(not alt):**

**tfidf=(1+ math.log(tf,10) ) \* (self.index.items[t].get('idf')) #((self.index.items[query\_tokens[temp2]].get('idf') )\* (1 + math.log( wordfreq[0] , 10)))**

**else:**

**n=self.index.nDocs**

**#print(self.index.items.get(t))**

**#print(type(self.index.items.get(t)))**

**df=len(list(self.index.items.get(t)['posting'].keys()))**

**idf\_k=math.log(((n-df)/df),10)**

**if(0<idf\_k):**

**tfidf=tf\*idf\_k**

**else:**

**tfidf=0**

**else:**

**tfidf=0**

**qcvector[t]=tfidf**

**# caculate the tf-idf for each document words and the then caculate the cosine similarity between document and query**

**finalvector={}**

**for doc in self.docs.docs:**

**dcvector = {}**

**title = doc.title**

**body = doc.body**

**# First convert both title and body words and lowercase**

**documentWords = title + body**

**#lowercasewords=ConvertLowerCase(doc)**

**#words=Tokenize(lowercasewords)**

**lowered\_documentWords = ConvertLowerCase(documentWords)**

**tokenizedWords = Tokenize(lowered\_documentWords)**

**removedstopwords = removeStopWords(tokenizedWords)**

**words = stemming(removedstopwords)**

**for w in words:**

**tf=1+ math.log(self.index.items[w].get('posting').get(doc.docID).get('term\_frequency'),10)**

**idf=self.index.items[w].get('idf')**

**tf\_idf=tf\*idf**

**dcvector[w]=tf\_idf**

**#print(dcvector)**

**sqrt\_sums\_dc=0**

**for x in dcvector:**

**square=dcvector[x] \* dcvector[x]**

**sqrt\_sums\_dc+=square**

**try:**

**reciprocalsqrt\_doc=(1/math.sqrt(sqrt\_sums\_dc))**

**except:**

**reciprocalsqrt\_doc=0**

**#caclulate d1/sqrt(d1^2) [sqrt(d1^2) is calculated above as reciprocalsqrt\_doc]**

**for x in dcvector:**

**dcvector[x] \*=reciprocalsqrt\_doc**

**#caculate cosine similarity between the Query words and docuemnt words**

**cosinevector=qcvector.copy();**

**i=0;**

**while i < qcvector.\_\_len\_\_():**

**qw=list(qcvector.keys())[i]**

**if(qw in dcvector ):**

**cosinevector[qw]=dcvector[qw]\*qcvector[qw]**

**else:**

**cosinevector[qw]=0**

**i=i+1**

**#find the sum of cosine value**

**sum\_cosine\_dc=0**

**for x in cosinevector:**

**sum\_cosine\_dc+=cosinevector[x]**

**finalvector[doc.docID]=sum\_cosine\_dc**

**#print(finalvector)**

**getTop3Results=[]**

**topthree = nlargest(k, finalvector, key=finalvector.get)**

**for key in topthree:**

**#print(key, ":", finalvector.get(key))**

**getTop3Results.append((key,finalvector.get(key)))**

**return getTop3Results**

**def BatchEvaluation(self):**

**TotalqueryTimeDic={}**

**number\_of\_queries = self.querynumber**

**docs = [int(x) for x in self.raw\_query.keys()]**

**for i in range(0,5):**

**query\_id\_list=random.sample(docs,number\_of\_queries)**

**print(query\_id\_list)**

**#booleanVector for each query:**

**booleanTime=0**

**for query\_id in query\_id\_list:**

**self.querynumber=query\_id**

**start = process\_time()**

**res=self.booleanQuery()**

**stop = process\_time()**

**booleanTime+=(stop-start)**

**TotalqueryTimeDic[str(i+1)]={"booleanModel":booleanTime}**

**vectorTime=0**

**for query\_id in query\_id\_list:**

**self.querynumber=query\_id**

**start=process\_time()**

**res=self.vectorQuery(3)**

**stop=process\_time()**

**vectorTime+=(stop-start)**

**TotalqueryTimeDic[str(i+1)].update({'vectorModel':vectorTime})**

**print(TotalqueryTimeDic)**

**with open('Processing\_time.csv', 'w') as f:**

**f.write("%s,%s,%s\n" % ("Iteration", "booleanModel(seconds)", "vectorModel(seconds)"))**

**for key in TotalqueryTimeDic.keys():**

**f.write("%s,%s,%s\n" % (key, TotalqueryTimeDic[key].get('booleanModel'), TotalqueryTimeDic[key].get('vectorModel')))**

**def test(obj):**

**''' test your code thoroughly. put the testing cases here'''**

**#Convert A sample Query to Terms**

**sample\_query="what similarity laws must be obeyed when constructing aeroelastic models of heated high speed aircraft ."**

**print("Sample Query Considered:: ",sample\_query)**

**lowered\_documentWords = ConvertLowerCase(sample\_query)**

**tokenizedWords = Tokenize(lowered\_documentWords)**

**spellcheckedwords = spellcheck(tokenizedWords)**

**removedstopwords = removeStopWords(spellcheckedwords)**

**stemmedWords = stemming(removedstopwords)**

**print("Sample Query Converted to Terms::",stemmedWords)**

**#run one boolean model for specific query and test the result**

**obj.querynumber=29**

**res=obj.booleanQuery()**

**assert res == ['462']**

**print("Test for Boolean Model Passed")**

**#run one vector model for specific query and test the result**

**res=obj.vectorQuery(3)**

**vector\_label = [x[0] for x in res]**

**assert vector\_label == ['462','1099','1340']**

**print('Test for Vector Model Passed')**

**def query(index\_file,model\_type,query\_file,query\_id):**

**''' 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**

**#load documents**

**inputdocument = cran.CranFile("cran.all")**

**#load the index file saved at from part 1**

**index=InvertedIndex().load(index\_file)**

**#load query processed files**

**queries=loadCranQry(query\_file)**

**qp=QueryProcessor(queries,index,inputdocument,query\_id)**

**if model\_type==0:**

**Booleanres=qp.booleanQuery()**

**print(Booleanres)**

**if model\_type==1:**

**vectorres=qp.vectorQuery(3,True)**

**print(vectorres)**

**if model\_type==2:**

**qp.BatchEvaluation()**

**#print("\*\*\*\*\*\*\*\*Running Test Cases \*\*\*\*\*\*\*\*\*\*\*\*")**

**#test(qp)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**#test()**

**#index\_file mode\_selection query.text qid\_or\_n**

**#query("index\_file",0,"query.text",53)**

**#query("index\_file",1,"query.text",4)**

**#query("index\_file", 2, "query.text", 2)**

**query(sys.argv[1], int(sys.argv[2]), sys.argv[3], int(sys.argv[4]))**

**batch\_eval.py**

**import scipy**

**from scipy import stats**

**import cran**

**import metrics**

**from cranqry import loadCranQry**

**from index import InvertedIndex**

**from query import \***

**def process\_querls\_file(qrels\_file,queries\_id\_list):**

**qrels\_dic={}**

**f=open(qrels\_file,'r').read()**

**row=f.split("\n")**

**#print(row)**

**for x in row:**

**if(x==''):**

**break**

**record=x.split(' ')**

**quer\_id=record[0]**

**doc\_id=record[1]**

**if(quer\_id in qrels\_dic.keys()):**

**qrels\_dic[quer\_id].append(doc\_id)**

**else:**

**qrels\_dic[quer\_id]=[doc\_id]**

**#print(qrels\_dic)**

**#ignore the query\_ids from querls.text which are not in query.text**

**#print(queries\_id\_list)**

**#print(qrels\_dic)**

**#print(len(qrels\_dic.keys()))**

**#print(queries\_id\_list.\_\_len\_\_())**

**temp\_dict = dict()**

**k=0**

**for x in qrels\_dic.keys():**

**newkey=queries\_id\_list[k]**

**temp\_dict[newkey] = qrels\_dic[x]**

**k+=1**

**#temp\_dict=dict(qrels\_dic)**

**# for x in temp\_dict.keys():**

**# if x not in queries\_id\_list:**

**# del qrels\_dic[x]**

**#print(temp\_dict)**

**return temp\_dict**

**def eval(index\_file,query\_file,qrels\_File,number\_of\_queries):**

**#read queryfile,indexfile**

**# ToDo**

**queries = loadCranQry(query\_file)**

**queries\_id\_list=[str(int(x)) for x in queries.keys()]**

**#print(queries\_id\_list)**

**#read querls.txt**

**qrels\_dict=process\_querls\_file(qrels\_File,queries\_id\_list)**

**inputdocument = cran.CranFile("cran.all")**

**# load the index file saved at from part 1**

**index = InvertedIndex().load(index\_file)**

**qp = QueryProcessor(queries, index, inputdocument, number\_of\_queries)**

**queries\_id\_list\_int=[int(x) for x in qrels\_dict.keys()]**

**queries\_id\_ls = [int(x) for x in queries.keys()]**

**#IdeaVectorsforQuery\_ids={}**

**sumbooleanNADC=[]**

**sumvectorNADC=[]**

**with open('Evaluation\_search.csv', 'w') as f:**

**f.write("%s,%s,%s,%s\n" % ("Iteration", "AverageNDCG-booleanModel", "AverageNDCG-vectorModel","P-value"))**

**for i in range(0,5):**

**vectorNADC=[]**

**booleanNADC=[]**

**intersection\_queries=list(set(queries\_id\_list\_int) & set(queries\_id\_ls))**

**random\_query\_id\_list = random.sample(queries\_id\_list\_int, number\_of\_queries)**

**#random\_query\_id\_list=[153, 18]**

**#print(random\_query\_id\_list)**

**for q\_id in random\_query\_id\_list:**

**print("Processing for Query ID ::",q\_id)**

**qp.querynumber=q\_id**

**#boolean\_res=qp.booleanQuery()**

**vector\_top3=qp.vectorQuery(5)**

**#vector\_top3=[('12',0.34),('746',0.33),('875',0.24)]**

**#print(boolean\_res)**

**print("Output for Vector Model Result::",vector\_top3)**

**if(vector\_top3.\_\_len\_\_()<1):**

**vectorNADC.append(0)**

**else:**

**vector\_label=[x[0] for x in vector\_top3]**

**score=[x[1] for x in vector\_top3]**

**print("DocumentIDs of Vector Model Result:: ",vector\_label)**

**print("Scores of Vector Model Result::",score)**

**true\_label=vector\_label.copy()**

**query\_id=str(q\_id)**

**for x in vector\_label:**

**#str\_x="{0:0=3d}".format(x)**

**ind=vector\_label.index(x)**

**if (x in qrels\_dict.get(query\_id)):**

**true\_label[ind]=1**

**else:**

**true\_label[ind]=0**

**if true\_label.\_\_len\_\_()<5:**

**len\_val=10-(true\_label.\_\_len\_\_())**

**true\_label.extend([0]\*len\_val)**

**print("Actual Vector:: ",true\_label)**

**print("Predicted Vector:: ",score)**

**if sum(true\_label)==0 :**

**vectorNADC.append(0)**

**else:**

**ndcg=metrics.ndcg\_score(true\_label, score,5)**

**print("Calculated ndcg for Vector::",ndcg)**

**vectorNADC.append(ndcg)**

**boolean\_res = qp.booleanQuery()**

**print("output of boolean\_res:: ",boolean\_res)**

**if boolean\_res.\_\_len\_\_()<1:**

**booleanNADC.append(0)**

**else:**

**score=[1]\*len(boolean\_res)**

**if(score.\_\_len\_\_()<5):**

**leng=5-(score.\_\_len\_\_())**

**score.extend([0]\*leng)**

**true\_label = boolean\_res.copy()**

**query\_id = str(q\_id)**

**for x in boolean\_res:**

**ind = boolean\_res.index(x)**

**if (x in qrels\_dict.get(query\_id)):**

**true\_label[ind] = 1**

**else:**

**true\_label[ind] = 0**

**if true\_label.\_\_len\_\_() < 5:**

**len\_val = 10 - (true\_label.\_\_len\_\_())**

**true\_label.extend([0] \* len\_val)**

**print("Actual boolean:: ", true\_label)**

**print("Predicted boolean:: ", score)**

**if sum(true\_label) == 0:**

**booleanNADC.append(0)**

**else:**

**ndcg = metrics.ndcg\_score(true\_label, score, 5)**

**print("Calculated ndcg for Boolean::", ndcg)**

**booleanNADC.append(ndcg)**

**print("Calculated NADC sum for all queries",vectorNADC)**

**avergae\_vectorNADC=float(sum(vectorNADC)/number\_of\_queries)**

**print("Calculated NADC sum for all queries",booleanNADC)**

**avergae\_booleanNADC=float(sum(booleanNADC)/number\_of\_queries)**

**print("Avergae NADC Vector::",avergae\_vectorNADC)**

**print("Avergae NADC boolean::",avergae\_booleanNADC)**

**p\_value=scipy.stats.wilcoxon(vectorNADC, booleanNADC, zero\_method='wilcox', correction=False)**

**print(i,str(avergae\_booleanNADC),str(avergae\_vectorNADC),str(p\_value[1]))**

**p="%.20f" % float(str(p\_value[1]))**

**print('P value for all the queries processed is:',p)**

**f.write("%s,%s,%s,%s\n" % (i+1, str(avergae\_booleanNADC), str(avergae\_vectorNADC),p))**

**print('Done')**

**if \_\_name\_\_ == '\_\_main\_\_':**

**#eval('index\_file', 'query.text', 'qrels.text', 50)**

**eval(str(sys.argv[1]), str(sys.argv[2]), str(sys.argv[3]), int(sys.argv[4]))**

**#eval()**

**test\_all.py**

**import cran**

**import query**

**from cranqry import loadCranQry**

**from index import InvertedIndex, test**

**from query import QueryProcessor**

**print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Test Cases Running for Index File\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**

**invertedobj=InvertedIndex()**

**test(invertedobj)**

**print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Test Cases Running for Query File\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**

**# load documents**

**inputdocument = cran.CranFile("cran.all")**

**# load the index file saved at from part 1**

**index = InvertedIndex().load("index\_file")**

**# load query processed files**

**queries = loadCranQry("query.text")**

**qp = QueryProcessor(queries, index, inputdocument, 29)**

**query.test(qp)**

**qp= QueryProcessor(queries,index,inputdocument,29)**

**qp.vectorQuery(3,True)**

**VectorModels.py**

**import scipy**

**from scipy import stats**

**import cran**

**import metrics**

**from batch\_eval import process\_querls\_file**

**from cranqry import loadCranQry**

**from index import InvertedIndex**

**from query import \***

**def VectorCompare():**

**queries = loadCranQry("query.text")**

**queries\_id\_list=[str(int(x)) for x in queries.keys()]**

**inputdocument = cran.CranFile("cran.all")**

**# load the index file saved at from part 1**

**index = InvertedIndex().load("index\_file")**

**qp = QueryProcessor(queries, index, inputdocument, 10)**

**queries\_id\_list=[str(int(x)) for x in queries.keys()]**

**#print(queries\_id\_list)**

**#read querls.txt**

**qrels\_dict=process\_querls\_file("qrels.text",queries\_id\_list)**

**#IdeaVectorsforQuery\_ids={}**

**sumbooleanNADC=[]**

**sumvectorNADC=[]**

**with open('Vector\_Evaluation\_search.csv', 'w') as f:**

**f.write("%s,%s,%s,%s\n" % ("Iteration", "AverageNDCG-booleanModel", "AverageNDCG-vectorModel","P-value"))**

**vectorNADC1 = []**

**booleanNADC2 = []**

**# random\_query\_id\_list=[153, 18]**

**# print(random\_query\_id\_list)**

**query\_id = [4 , 29, 53, 58, 100]**

**vectorNADC1=[]**

**vectorNADC2=[]**

**for q\_id in query\_id:**

**qp.querynumber = q\_id**

**# boolean\_res=qp.booleanQuery()**

**vector\_top3 = qp.vectorQuery(5)**

**vector2\_top3=qp.vectorQuery(5,True)**

**# vector\_top3=[('12',0.34),('746',0.33),('875',0.24)]**

**# print(boolean\_res)**

**print("Output for Vector Model Result::", vector\_top3)**

**if (vector\_top3.\_\_len\_\_() < 1):**

**vectorNADC1.append(0)**

**else:**

**vector\_label = [x[0] for x in vector\_top3]**

**score = [x[1] for x in vector\_top3]**

**print("DocumentIDs of Vector Model Result:: ", vector\_label)**

**print("Scores of Vector Model Result::", score)**

**true\_label = vector\_label.copy()**

**query\_id = str(q\_id)**

**for x in vector\_label:**

**# str\_x="{0:0=3d}".format(x)**

**ind = vector\_label.index(x)**

**if (x in qrels\_dict.get(query\_id)):**

**true\_label[ind] = 1**

**else:**

**true\_label[ind] = 0**

**if true\_label.\_\_len\_\_() < 5:**

**len\_val = 10 - (true\_label.\_\_len\_\_())**

**true\_label.extend([0] \* len\_val)**

**print("Actual Vector:: ", true\_label)**

**print("Predicted Vector:: ", score)**

**if sum(true\_label) == 0:**

**vectorNADC1.append(0)**

**else:**

**ndcg = metrics.ndcg\_score(true\_label, score, 5)**

**print("Calculated ndcg for Vector::", ndcg)**

**vectorNADC1.append(ndcg)**

**if (vector2\_top3.\_\_len\_\_() < 1):**

**vectorNADC2.append(0)**

**else:**

**vector\_label = [x[0] for x in vector2\_top3]**

**score = [x[1] for x in vector2\_top3]**

**print("DocumentIDs of Vector Model Result:: ", vector\_label)**

**print("Scores of Vector Model Result::", score)**

**true\_label = vector\_label.copy()**

**query\_id = str(q\_id)**

**for x in vector\_label:**

**# str\_x="{0:0=3d}".format(x)**

**ind = vector\_label.index(x)**

**if (x in qrels\_dict.get(query\_id)):**

**true\_label[ind] = 1**

**else:**

**true\_label[ind] = 0**

**if true\_label.\_\_len\_\_() < 5:**

**len\_val = 10 - (true\_label.\_\_len\_\_())**

**true\_label.extend([0] \* len\_val)**

**print("Actual Vector:: ", true\_label)**

**print("Predicted Vector:: ", score)**

**if sum(true\_label) == 0:**

**vectorNADC2.append(0)**

**else:**

**ndcg = metrics.ndcg\_score(true\_label, score, 5)**

**print("Calculated ndcg for Vector::", ndcg)**

**vectorNADC2.append(ndcg)**

**print("Calculated NADC sum for all queries", vectorNADC1)**

**avergae\_vectorNADC = float(sum(vectorNADC1) / 5)**

**print("Calculated NADC sum for all queries", vectorNADC2)**

**avergae\_vectorNADC2 = float(sum(vectorNADC2) / 5)**

**print("Avergae NADC Vector::", avergae\_vectorNADC)**

**print("Avergae NADC boolean::", avergae\_vectorNADC2)**

**print(vectorNADC1)**

**print(vectorNADC2)**

**p\_value = scipy.stats.wilcoxon(vectorNADC1, vectorNADC2, zero\_method='wilcox', correction=False)**

**p = "%.20f" % float(str(p\_value[1]))**

**print('P value for all the queries processed is:', p)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**VectorCompare()**