**CS 7800 Information Retrieval**

**Simple Search Engine**

**Abhishek Singh Thakur**

**(U00933835)**

**Code Snippets**

**Processing the text in util.py**: Text would be processed in order to achieve tokenization, lower case, removal of stop words and stemming

import nltk

from nltk.stem.porter import \*

import re

import norvig\_spell

from norvig\_spell import \*

def tokenizer(text):

    text = re.sub("[^a-zA-Z]+", " ", text)

    tokens = nltk.tokenize.word\_tokenize(text)

    return tokens

# ''' 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'''

# 1. Convert to lower cases,

# 2. Remove stopwords,

# 3. Stemming

def preprocessing\_txt(text, query):

    tokens = text.split()

    # tokens = tokenizer(text)

    stemmer = PorterStemmer()

    processedText = ""

    removedStopwords = []

    stemmedWords = []

    if query == 'true':

        for i in range(len(tokens)):

            tokens[i] = correction(tokens[i])

    for token in tokens:

        token = token.lower()

        if token not in open('stopwords').read():

            processedText += stemmer.stem(token)

            processedText += " "

            if query == 'test':

                stemmedWords.append(stemmer.stem(token))

        elif query == 'test':

            removedStopwords.append(token)

    if query == 'test':

        print('Removed stop words are:' + str(removedStopwords))

        print('Stemmed words are:     ' + str(stemmedWords))

    return processedText

def isStopWord(word):

    n = 0

    for i in word.split():

        if i not in open('stopwords').read():

            n += 1

            if n == len(word.split()) - 1:

                print('No stop words founds')

**Inverted Index creation Code implementation in -- index.py**

# Indexing the Cranfield dataset and save the index to a file

# The index is saved to index\_file and TFIDF scores are stored in tfidf scores for the terms in the documents from cran.all

def indexingCranfield(cranfile,savefile, test):

    # command line usage: "python index.py cran.all index\_file"

    file = CranFile(cranfile)

    invertedIndex = InvertedIndex()

    temp = {}

    termFrequency = {}

    n = 0

    totalDocs = len(file.docs)

    print(str(len(file.docs)) + ' documents are present in the dataset.')

    for i in file.docs:

        if n == totalDocs: break

        else:

            # For calculating the Term frequency according to the documents

            temp = calculateTF(preprocessing\_txt(i.body,''))

            termFrequency[i.docID] = temp

            # For creating the index\_file

            invertedIndex.indexDoc(i)

            n += 1

    invertedIndex.save(savefile, termFrequency, totalDocs, test)

    print ('Indexing file creation  is done')

class InvertedIndex:

    inverted\_index = {}

    inverted\_index\_temp = {}

    def \_\_init\_\_(self):

        self.items = {} # list of IndexItems

        self.nDocs = 0  # the number of indexed documents

    def indexDoc(self, doc): # indexing a Document object using functions in util.py

        processedText = preprocessing\_txt(doc.body,'')

        for word in processedText.split():

            position = wordPositions(word, processedText)

            if word in self.items.keys():

                if not int(doc.docID) in list(self.items[word].posting.keys()):

                    self.items[word].add(int(doc.docID), position)

            else:

                index\_item = IndexItem(word)

                index\_item.add(int(doc.docID), position)

                self.items[word] = index\_item

        for i in self.items:

            dictionary = {}

            x = 0

            for j in self.items[i].posting:

                listposition = []

                for k in self.items[i].posting[j]:

                    listposition.append(k)

                dictionary[j] = listposition

            self.inverted\_index\_temp[i] = dictionary

        self.inverted\_index = self.inverted\_index\_temp

# To find the normalized term location in the processed query

def wordPositions(word, processedText):

    pos = 1

    positions = []

    for data in processedText.split():

        if word == data:

            positions.append(pos)

        pos += 1

    return positions

Calculating TF-IDF for the document set cran.all which is implemented in index.py

def calculateIDF(doc, indexData, totalDocs):

    for i in doc.keys():

        if i in indexData.keys():

            count = len(indexData[i].keys())

            doc[i] = doc[i] \* float( 1 + math.log10(totalDocs/count))

    return doc

# For calculating the normalized Term Frequency of the terms in the query

def calculateTF(text):

    textList = text.split()

    count = 0

    textTF = {}

    for i in range(len(textList)):

        if textList[i] in textTF.keys():

            textTF[textList[i]] += float(1/len(textList))

        else:

            textTF[textList[i]] = float(1/len(textList))

    return textTF

Saving Inverted Index in index\_file and TF-IDF scores in the tfidf file for future use while calculating cosine similarity and relevant documents when using Boolean and Vector model for a sample query.

# Calculating and saving to disk TFIDF for the documents in the cran.all and creating, serialize/deserialize the index indexing for the terms from all docs with their positions

    def save(self, savefile, termFrequency, totalDocs, test):

        for i in termFrequency:

            termFrequency[i] = calculateIDF(termFrequency[i], self.inverted\_index, totalDocs)

        # Storing tfidf scores for calculating Cosine similarity

        if test == 'true':

            with open('test\_tfidf','w') as data:

                json.dump(termFrequency, data)

            with open('test\_index','w') as fp:

                json.dump(self.inverted\_index, fp)

        else:

            with open('tfidf','w') as data:

                json.dump(termFrequency, data)

            with open(savefile,'w') as fp:

                json.dump(self.inverted\_index, fp)

Selecting of the Modes based on the command line input:

    if mode == 0:

        booleanQuery(data, query, '')

    elif mode == 1:

        vectorQuery(data, query, tfidf,'', 1400)

    else:

        startBool = time.process\_time()

        for i in sampleQueries:

            booleanQuery(data, i, 'time')

        print('Boolean Time         ' + str(time.process\_time() - startBool))

        startVector = time.process\_time()

        for i in sampleQueries:

           vectorQuery(data, i, tfidf, 'time',1400)

        print('Vector Time          ' + str(time.process\_time() - startVector))

**Boolean Model Code Implementation: query.py**

# 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.

# Return a list of docIDs which matches with the query i.e. relavant documents from cran.all

def booleanQuery(self, query, eval):

        processedText = preprocessing\_txt(query, 'true')

        if eval == 'test':

            print('For making sure queries are converted to terms and the length of the termslist is    ' + str(len(processedText.split())))

        queryDocs = []

        temp = []

        ndcgBoolean = []

        n = 0

        for i in processedText.split():

            if i in self:

                queryDocs.append(list(self[i].keys()))

        for i in range(len(queryDocs)):

            if i == 0:

                temp = queryDocs[i]

            else:

                temp = commonDocs(queryDocs[i],  temp)

        if eval == 'boolean':

            if len(temp) == 0:

                noDocs = [0,0,0,0,0]

                return noDocs

            else:

                for i in range(5):

                    if i < len(temp):

                        ndcgBoolean.append(temp[i])

                    else:

                        ndcgBoolean.append(0)

                return ndcgBoolean

            print(ndcgBoolean)

        elif eval == 'time':

            return

        elif eval == 'test':

            if len(temp) > 0:

                print("The actual document which contains the query are --" + str(temp)[1:-1])

        else:

            if len(temp) == 0:

                print('No Matching Results found')

            else:

                print("Boolean Model -- " + str(len(temp)) + " documents which contains the query are " + str(temp)[1:-1])

            return

# To find the common documents between two documents while calculating results for boolean model

def commonDocs(a, b):

    docs=[]

    for i in range(len(a)):

        for j in range(len(b)):

            if a[i] == b[j]:

                docs.append(a[i])

                break

    return docs

**Vector Model implementation: query.py**

# For vectorQuery, the program will output the top 3 most similar documents for a query using the cosine similarity

# Returns top k pairs of (docID, similarity), ranked by their cosine similarity with the query in the descending order

def vectorQuery(self, query, tfidf, eval, totalDocs):

    queryTerms = preprocessing\_txt(query, 'true')

    if eval == 'test':

        print('For making sure queries in Vector Modelare converted to terms and the length of the termslist is' + str(len(queryTerms.split())))

    tffidfQuery, tffidfIndex, tfidfDocs, docTF = {}, {}, {}, {}

    # totalDocs = 1400

    docSimilarity = []

    tffidfQuery = tffIdfQuery(queryTerms.split(), self, totalDocs)

    tfidfIndex = tfIdfIndex(queryTerms.split(), self, tfidf, totalDocs)

    for doc in range(totalDocs):

        for i in queryTerms.split():

            tfidfDocs[i] = {}

            tfidfDocs[i] = tfidfIndex[i][str(doc+1)]

        temp = cosineSimilarity(tffidfQuery, tfidfDocs)

        docTF[temp] = {}

        docTF[temp] = str(doc + 1)

        docSimilarity.append(temp)

    docSimilarity = sorted(docSimilarity, key = lambda x:float(x))

    if eval == 'vector':

        ndcgVector = [0,0,0,0,0]

        cosine\_value = [0,0,0,0,0]

        for i in range(5):

                if i < len(docSimilarity):

                    ndcgVector[i] = docTF[docSimilarity[totalDocs-(i+1)]]

                    cosine\_value[i] = docSimilarity[totalDocs-(i+1)]

                else:

                    ndcgVector[i] = 0

                    cosine\_value[i] = 0.0

        return ndcgVector+cosine\_value

    elif eval == 'time':

        return

    else:

        print('Vector Model -- Top 3 ranked Documents are ' + docTF[docSimilarity[totalDocs-1]] +', ' +

         docTF[docSimilarity[totalDocs-2]] +', ' + docTF[docSimilarity[totalDocs-3]])

        print('                Their scores are           ' + str(docSimilarity[totalDocs-1]) +', ' + str(docSimilarity[totalDocs-2]) +', ' + str(docSimilarity[totalDocs-3]))

# Calculating cosine similarity for ranking the documents against documents

def cosineSimilarity(querytfidf, doctfidf):

    modQuery = 0

    modDoc = 0

    dotProduct = 0

    for i in querytfidf.keys():

        modQuery += (querytfidf[i] \* querytfidf[i])

        modDoc += (doctfidf[i] \* doctfidf[i])

        dotProduct += (querytfidf[i] \* doctfidf[i])

    if modQuery\*modDoc == 0 :

        modDoc = 0.00000000000001

        modQuery = 0.0000000000001

    return float(dotProduct/(math.sqrt(modQuery) \* math.sqrt(modDoc)))

Calculating TFIDF scores for the query sample, fetching the documents with the terms in the query sample:

# To calculate TDIDF scores for the query sample.

def tffIdfQuery(queryList, self, totalDocs):

    queryDict = {}

    for term in queryList:

        if term in queryDict.keys():

            if term in self:

                queryDict[term] += ((1/len(queryList)) \* (1 + math.log10(totalDocs/len(self[term].keys()))))

            else:

                queryDict[term] += ((1/len(queryList)) \* (1 + math.log10(1)))

        else:

            queryDict[term] = {}

            if term in self:

                queryDict[term] = ((1/len(queryList)) \* (1 + math.log10(totalDocs/len(self[term].keys()))))

            else:

                queryDict[term] = ((1/len(queryList)) \* (1 + math.log10(1)))

    return queryDict

# Fetching the documents with the TFIDF scores stored in tfidf file for every token in the query sample

def tfIdfIndex(queryList, self, tf, totalDocs):

    dict = {}

    for i in queryList:

        dict[i] = {}

        if i in self:

            # print(str(self[i].keys()))

            for k in range(totalDocs):

                if str(k+1) in self[i].keys():

                   dict[i][str(k+1)] = 0

                   dict[i][str(k+1)] = tf[str(k+1)][i]

                else:

                   dict[i][str(k+1)] = 0

        else:

            for m in range(totalDocs):

                dict[i][str(m+1)] = 0

    return dict

Mapping the Query samples from query.text with the qrels.text since docID’s are not continues in the query.text for easy evaluation of Boolean and vector model results against the actual results. **batch\_eval.py**

sampleQueries = []

    f = open(sys.argv[3])

    file = loadCranQry(sys.argv[2])

    with open(sys.argv[1]) as fp:

            data = json.load(fp)

    with open('tfidf') as tfidfFile:

            tfidf = json.load(tfidfFile)

    queryIds = list(file.keys())

    count = int(sys.argv[4])

    querySamples = random.choices(queryIds, k=count)

    querySamples = list(set(querySamples))

    print(querySamples)

    for i in querySamples:

        sampleQueries.append(file[i].text)

    querydict = {}

    mappingDict = {}

    for i in f:

        j = i.split()

        if j[0] in querydict.keys():

            querydict[j[0]].append(j[1])

        else:

            querydict[j[0]] = []

            querydict[j[0]].append(j[1])

    n = 1

    for i in file.keys():

        mappingDict[i] = []

        mappingDict[i] = querydict[str(n)]

        n += 1

**Calculating NDGC\_5 scores and the Wilcoxon results for evaluating Boolean Model and Vector Model results with the actual results: batch\_eval.py**

import metrics

from cranqry import \*

import random

import json

from query import \*

from metrics import \*

from scipy import stats

# Evaluating the NDCG scores for Boolean model and Vector model results against actual data to check the success rate.

# Prints average NDCG scores for the boolean model, Vector model  and p\_value score for sample of queries of size k

def eval():

    queryMatchesB = []

    queryMatchesV = []

    ndcg\_scoreB = []

    ndcg\_scoreV = []

    for i in range(len(sampleQueries)):

        queryMatchesB = booleanQuery(data, sampleQueries[i], 'boolean')

        ndcg\_scoreB.append(calculate\_ndgc5(queryMatchesB, mappingDict[querySamples[i]],'boolean'))

        queryMatchesV = vectorQuery(data, sampleQueries[i], tfidf, 'vector', 1400)

        ndcg\_scoreV.append(calculate\_ndgc5(queryMatchesV, mappingDict[querySamples[i]],'vector'))

    avgBoolean = sum(ndcg\_scoreB) / len(ndcg\_scoreB)

    avgVector = sum(ndcg\_scoreV) / len(ndcg\_scoreV)

    p\_value = stats.wilcoxon(ndcg\_scoreB,ndcg\_scoreV)

    print('Average NDCG score for Boolean Model is  ' + str(avgBoolean))

    print('Average NDCG score for Vector Model is   ' + str(avgVector))

    print('P-Value calculated for Boolean Vector is ' + str(p\_value))

# Calculating NDCG scores for first 5 results for boolean and vector model.

def calculate\_ndgc5(docId,resultedDocs, method):

    y\_truth = [0,0,0,0,0]

    y\_score = [0,0,0,0,0]

    for i in range(5):

        if docId[i] == 0:

            y\_truth[i] = 0

            y\_score[i] = 0

        else:

            if method == 'boolean':

                y\_score[i] = 1

            else:

                y\_score[i] = docId[5+i]

            if docId[i] in resultedDocs:

                y\_truth[i] = 1

            else:

                y\_truth[i] = 0

    ndcgValue = ndcg\_score(y\_truth,y\_score)

    return ndcgValue