CS6370: Natural Language Processing
Project

Release Date: 21st March 2024 Deadline:

Name:	Roll No.:
Bapan Mandal	CS21B016
Ujjayan Pal	CS21B084
Abhishek Mahajan	CS23M002

General Instructions:

1. The template for the code (in Python) is provided in a separate zip file. You are expected to fill in the template wherever instructed. Note that any Python library, such as nltk, stanfordcorenlp, spacy, etc, can be used.
2. A folder named 'Roll_number.zip' that contains a zip of the code folder and your responses to the questions (a PDF of this document with the solutions written in the text boxes) must be uploaded on Moodle by the deadline.
3. Any submissions made after the deadline will not be graded.
4. Answer the theoretical questions concisely. All the codes should contain proper comments.
5. For questions involving coding components, paste a screenshot of the code.
6. The institute's academic code of conduct will be strictly enforced.

The first assignment in the NLP course involved building a basic text processing module that implements sentence segmentation, tokenization, stemming / lemmatization, stopword removal, and some aspects of spell check. This module involves implementing an Information Retrieval system using the Vector Space Model. The same dataset as in Part 1 (Cranfield dataset) will be used for this purpose. The project is split into two components - the first is a *warm-up* component comprising of Parts 1 through 4 that would act as a precursor for the second and main component, where you improve over the basic IR system.

[Warm up] Part 1: Working out a toy IR system [Numerical]

Consider the following three documents:

d_1: Herbivores are typically plant eaters and not meat eaters
d_2: Carnivores are typically meat eaters and not plant eaters
d_3: Deers eat grass and leaves

1. Assuming {are, and, not} as stop words, arrive at an inverted index representation for the above documents.

Herbivores → d_1 typically → d_1, d_2 plant → d_1, d_2 eaters → d_1, d_2 meat → d_1, d_2 Carnivores → d_2 Deers → d_3 eat → d_3 grass → d_3 leaves → d_3

2. Construct the TF-IDF term-document matrix for the corpus $\{d_1, d_2, d_3\}$.

	Counts, tf_i				Weights, $w_i = tf_i \times IDF_i$		
Terms	d_1	d_2	d_3	$IDF_i = \log_{10}(D/df_i)$	d_1	d_2	d_3
Herbivore	1	0	0	0.4771	0.4771	0	0
typically	1	1	0	0.1761	0.1761	0.1761	0
plant	1	1	0	0.1761	0.1761	0.1761	0
eaters	2	2	0	0.1761	0.3522	0.3522	0
meat	1	1	0	0.1761	0.1761	0.1761	0
Carnivore	0	1	0	0.4771	0	0.4771	0
Deers	0	0	1	0.4771	0	0	0.4771
eat	0	0	1	0.4771	0	0	0.4771
grass	0	0	1	0.4771	0	0	0.4771
leaves	0	0	1	0.4771	0	0	0.4771

3. Suppose the query is "plant eaters," which documents would be retrieved based on the inverted index constructed before?

Documents: d_1, d_2

4. Find the cosine similarity between the query and each of the retrieved documents. Is the result desirable? Why?

Cosine Similarity calculations:
$Q = \{0, 0, 0.1761, 0.1761, 0, 0, 0, 0, 0, 0\}$
$|Q| = 0.2490$
$|D_1| = 0.6669$
$|D_2| = 0.6669$
$|D_3| = 0.9542$
$Q \cdot D_1 = 0.0930$
$Q \cdot D_2 = 0.0930$
$Q \cdot D_3 = 0$

$sim(Q, D_1) = 0.56$
$sim(Q, D_2) = 0.56$
$sim(Q, D_3) = 0$

Ranking documents:
D_1/D_2 then followed by D_3 (tie between first two documents)

Is the ordering desirable? If no, why not?:
No. We are unable to determine which document to be prioritised among D_1 and D_2

[Warm up] Part 2: Building an IR system [Implementation]

1. Implement the retrieval component of the IR system in the template provided. Use the TF-IDF vector representation for representing documents.

```python
import numpy as np
from scipy import spatial

class LSA():

    def reduced_tfidf(self, tfidf):
        """
        Parameters
        ----------
        arg1 : matrix
            The tfidf matrix of all the docs with all the terms in the corpus
        Returns
        -------
            The reduced tfidf matrix and the matrices obtained after SVD and reducing
the dimensionality
        """
        tfidf = np.transpose(tfidf)
        len_corpus = np.shape(tfidf)[0]
        len_docs = np.shape(tfidf)[1]

        u,s,vt = np.linalg.svd(tfidf); # u: txt, s: txd, vt = dxd
        s_size = np.size(s) # 1400 = d
        s_values = np.zeros([len_corpus, len_docs]) # txd
        s_values[:s_size, :s_size] = np.diag(s) # dxd is diagonal
        us = np.dot(u,s_values) # txd
        tfidf1 = np.dot(us, vt) # txd

        totalSum = s.sum()
        currSum = 0
        count = 0
        for elem in s:
            if (currSum/totalSum) > 0.65:
                break
            else:
                currSum = currSum + elem
                count = count + 1
        #print(count)
        s_values_k = s_values[:count, :count] # sxs
        u_k = u[:, :count] # txs
        vt_k = vt[:count, :] # sxd
        us_k = np.dot(u_k,s_values_k) # txs
        tfidf_k = np.dot(us_k, vt_k) # txd

        return tfidf_k, u_k, s_values_k, vt_k

    def cosine_similarity(self, T, S, D, tfidf_query):
        """
        Parameters
        ----------
        arg1 : matrix
            Terms in concept space, after SVD and reducing dimensionality
        arg2 : matrix
            Singular values after SVD on tf-idf matrix
        arg3 : matrix
            Docs in concept space, after SVD and reducing dimensionality
        arg4 : vector
            The tfidf vector of a query with all the terms in the corpus
        Returns
        -------
            The cosine similarities of the query with all the documents in the dataset
        """
        DS_docs_matrix = np.dot(D, S) # dxs
        DS_query = np.dot(tfidf_query,T) # 1xs

        cosine_sims = []
        for DS_doc in DS_docs_matrix: # 1xs
            cosine_sim = spatial.distance.cosine(DS_query, DS_doc)
            '''
            if np.linalg.norm(DS_doc) == 0:
                cosine_sim = 0
            else:
                cosine_sim = np.dot(DS_query, DS_doc)/((np.linalg.norm(DS_query)) *
(np.linalg.norm(DS_doc)))
            '''
            cosine_sims.append(cosine_sim)

        return cosine_sims
```

```python
from util import *

# Add your import statements here
import numpy as np
from scipy import spatial
from math import log
from collections import Counter

from lsa import LSA

def decapitalize(s, upper_rest = False):
  return ''.join([s[:1].lower(), (s[1:].upper() if upper_rest else s[1:])])

class InformationRetrieval():

    def __init__(self):
        self.index = None
        self.N = 0
        self.idf = {}
        self.docVecs = {}
        self.lsa = LSA()

    def buildIndex(self, docs, docIDs):
        """
        Builds the document index in terms of the document
        IDs and stores it in the 'index' class variable

        Parameters
        ----------
        arg1 : list
            A list of lists of lists where each sub-list is
            a document and each sub-sub-list is a sentence of the document
        arg2 : list
            A list of integers denoting IDs of the documents
        Returns
        -------
        None
        """

        index = {}

        #Fill in code here

        N = len(docIDs)
        self.N = N
        for i in range(N):
            for sentence in docs[i]:
                for word in sentence:
                    if word in index:
                        index[word].append(docIDs[i])
                    else:
                        index[word] = [docIDs[i]]

        self.index = index
        for t in index:
            index[t] = Counter(index[t])
            nt = len(index[t].keys())
            self.idf[t] = log(N/nt)

        docVec = {}

        for i in range(N):
            if len(docs[i])==0:
                vec = [0]*len(index)
            else:
                vec = []
                for t in index:
                    #nt = len(index[t].keys())
                    #print(len(docs[i]))
                    vec.append(index[t][docIDs[i]]*self.idf[t]/len(docs[i]))
            self.docVecs[docIDs[i]] = (vec)


    def rank(self, queries):
        """
        Rank the documents according to relevance for each query

        Parameters
        ----------
        arg1 : list
            A list of lists of lists where each sub-list is a query and
            each sub-sub-list is a sentence of the query


        Returns
        -------
        list
            A list of lists of integers where the ith sub-list is a list of IDs
            of documents in their predicted order of relevance to the ith query
        """

        doc_IDs_ordered = []

        #Fill in code here
        for query in queries:

            Qvec = self.convertQuery2Vector(query)
            CosSim = {}
            '''
            Qvec = []
            #print(query)
            c = {}
            for sentence in query:
                sentence = [decapitalize(str(q)) for q in sentence]
                for word in sentence:
                    if word in c:
                        c[word]+=1
                    else:
                        c[word] = 1

            for t in self.index:
                nt = len(self.index[t].keys())
                try:
                    Qvec.append(c[t]*self.idf[t]/len(query))
                except:
                    Qvec.append(0)
            '''

            for docID in self.docVecs:
                CosSim[docID] = spatial.distance.cosine(Qvec,self.docVecs[docID])


            doc_IDs_ordered.append(sorted(CosSim, key=CosSim.get, reverse = False)  )


        return doc_IDs_ordered

    def rankUsingLSA(self, docs, docIDs,queries):

        tfidf = []
        for docId in self.docVecs:
            tfidf.append(self.docVecs[docId])

        '''
        inv_index, corpus = self.utilities.inverted_index(docs)

        # idf
        idf = {}
        for word in corpus:
            idf[word] = np.log10(len(docs)/(len(inv_index[word])))

        # tfidf
        tfidf = np.zeros([len(docs),len(corpus)])
        doc_index = 0
        for doc in docs:
            for sentence in doc:
                for word in sentence:
                    if word in corpus:
                        word_index = corpus.index(word)
                        tfidf[doc_index][word_index] += idf[word] # multiple adds -> covers tf
            doc_index += 1
        '''

        corpus = list(self.index.keys())

        tfidf_k, u_k, s_values_k, vt_k = self.lsa.reduced_tfidf(tfidf)
        index = {
            "corpus" : corpus,
            "idf" : self.idf,
            "tfidf" : tfidf_k,
            "T": u_k, # txs
            "S": s_values_k, # sxs
            "D": np.transpose(vt_k) # dxs
        }
        #self.index = index

        doc_IDs_ordered = []
        # class variables
        corpus = index["corpus"]
        idf = index["idf"]
        tfidf_docs = index["tfidf"]

        tfidf = np.zeros([len(queries), len(corpus)])
        query_index = 0
        all_cosine_sims = []
        # tfidf for queries
        for query in queries:
            '''
            for sentence in query:
                for word in sentence:
                    word = word.lower()
                    if word != '.' and word in corpus:
                        word_index = corpus.index(word)
                        tfidf[query_index][word_index] += idf[word] # multiple adds -> covers tf

            query_tfidf = tfidf[query_index]
            '''
            query_tfidf = np.array(self.convertQuery2Vector(query))

            T = index["T"] # txs
            S = index["S"] # sxs
            D = index["D"] # dxs
            cosine_sims = self.lsa.cosine_similarity(T,S,D,query_tfidf)
            all_cosine_sims.append(cosine_sims)
            query_index += 1
        # ranking docs
        docIDs = [i+1 for i in range(len(tfidf_docs))]
        for i in all_cosine_sims:
            sorted_sim = np.argsort(i)
            doc_IDs_ordered.append([docIDs[j] for j in sorted_sim])

        return doc_IDs_ordered

    def convertQuery2Vector(self, query):
        Qvec = []
        CosSim = {}
        #print(query)
        c = {}
        for sentence in query:
            sentence = [decapitalize(str(q)) for q in sentence]
            for word in sentence:
                if word in c:
                    c[word]+=1
                else:
                    c[word] = 1

        for t in self.index:
            nt = len(self.index[t].keys())
            try:
                Qvec.append(c[t]*self.idf[t]/len(query))
            except:
                Qvec.append(0)
        return Qvec
```

[Warm up] Part 3: Evaluating your IR system [Implementation]

1. Implement the following evaluation measures in the template provided (i). Precision@k, (ii). Recall@k, (iii). $F_{0.5}$ score@k, (iv). AP@k, and (v) nDCG@k.

Precision@k:

```
def queryPrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    """
    Computation of precision of the Information Retrieval System
    at a given value of k for a single query

    Parameters
    ----------
    arg1 : list
        A list of integers denoting the IDs of documents in
        their predicted order of relevance to a query
    arg2 : int
        The ID of the query in question
    arg3 : list
        The list of IDs of documents relevant to the query (ground truth)
    arg4 : int
        The k value

    Returns
    -------
    float
        The precision value as a number between 0 and 1
    """

    precision = -1

    #Fill in code here

    retAndRelevant = list(set(query_doc_IDs_ordered[:k]) & set(true_doc_IDs))
    precision = len(retAndRelevant) / k
    return precision


def meanPrecision(self, doc_IDs_ordered, query_ids, qrels, k):
    """
    Computation of precision of the Information Retrieval System
    at a given value of k, averaged over all the queries

    Parameters
    ----------
    arg1 : list
        A list of lists of integers where the ith sub-list is a list of IDs
        of documents in their predicted order of relevance to the ith query
    arg2 : list
        A list of IDs of the queries for which the documents are ordered
    arg3 : list
        A list of dictionaries containing document-relevance
        judgements - Refer cran_qrels.json for the structure of each
        dictionary
    arg4 : int
        The k value

    Returns
    -------
    float
        The mean precision value as a number between 0 and 1
    """

    meanPrecision = -1

    #Fill in code here

    totalPrecision = 0
    qRelsDict = {}
    for qRel in qrels:
        if qRel["query_num"] in qRelsDict:
            qRelsDict[qRel["query_num"]].append(int(qRel["id"]))
        else:
            qRelsDict[qRel["query_num"]] = [int(qRel["id"])]

    for i in range(0, len(query_ids)):
        totalPrecision = totalPrecision + self.queryPrecision(doc_IDs_ordered[i], query_ids[i], list(qRelsDict[str(query_ids[i])]), k)
    meanPrecision = totalPrecision / len(query_ids)
    return meanPrecision
```

Recall@k:

```python
    def queryRecall(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        """
        Computation of recall of the Information Retrieval System
        at a given value of k for a single query

        Parameters
        ----------
        arg1 : list
            A list of integers denoting the IDs of documents in
            their predicted order of relevance to a query
        arg2 : int
            The ID of the query in question
        arg3 : list
            The list of IDs of documents relevant to the query (ground truth)
        arg4 : int
            The k value

        Returns
        -------
        float
            The recall value as a number between 0 and 1
        """

        recall = -1

        #Fill in code here
        retAndRelevant = list(set(query_doc_IDs_ordered[:k]) & set(true_doc_IDs))
        recall = len(retAndRelevant) / len(true_doc_IDs)
        return recall


    def meanRecall(self, doc_IDs_ordered, query_ids, qrels, k):
        """
        Computation of recall of the Information Retrieval System
        at a given value of k, averaged over all the queries

        Parameters
        ----------
        arg1 : list
            A list of lists of integers where the ith sub-list is a list of IDs
            of documents in their predicted order of relevance to the ith query
        arg2 : list
            A list of IDs of the queries for which the documents are ordered
        arg3 : list
            A list of dictionaries containing document-relevance
            judgements - Refer cran_qrels.json for the structure of each
            dictionary
        arg4 : int
            The k value

        Returns
        -------
        float
            The mean recall value as a number between 0 and 1
        """

        meanRecall = -1

        #Fill in code here

        totalRecall = 0
        qRelsDict = {}
        for qRel in qrels:
            if qRel["query_num"] in qRelsDict:
                qRelsDict[qRel["query_num"]].append(int(qRel["id"]))
            else:
                qRelsDict[qRel["query_num"]] = [int(qRel["id"])]

        for i in range(0, len(query_ids)):
            totalRecall = totalRecall + self.queryRecall(doc_IDs_ordered[i], query_ids[i], list(qRelsDict[str(query_ids[i])]), k)
        meanRecall = totalRecall / len(query_ids)
        return meanRecall
```

$F_{0.5}$ score@k:

```python
def queryFscore(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    """
    Computation of fscore of the Information Retrieval System
    at a given value of k for a single query

    Parameters
    ----------
    arg1 : list
        A list of integers denoting the IDs of documents in
        their predicted order of relevance to a query
    arg2 : int
        The ID of the query in question
    arg3 : list
        The list of IDs of documents relevant to the query (ground truth)
    arg4 : int
        The k value

    Returns
    -------
    float
        The fscore value as a number between 0 and 1
    """

    fscore = -1

    #Fill in code here

    precision = self.queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    recall = self.queryRecall(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    if precision == 0 and recall == 0:
        return 0
    fscore = (2*precision*recall)/(precision+recall)
    return fscore


def meanFscore(self, doc_IDs_ordered, query_ids, qrels, k):
    """
    Computation of fscore of the Information Retrieval System
    at a given value of k, averaged over all the queries

    Parameters
    ----------
    arg1 : list
        A list of lists of integers where the ith sub-list is a list of IDs
        of documents in their predicted order of relevance to the ith query
    arg2 : list
        A list of IDs of the queries for which the documents are ordered
    arg3 : list
        A list of dictionaries containing document-relevance
        judgements - Refer cran_qrels.json for the structure of each
        dictionary
    arg4 : int
        The k value

    Returns
    -------
    float
        The mean fscore value as a number between 0 and 1
    """

    meanFscore = -1

    #Fill in code here

    totalFScore = 0
    qRelsDict = {}
    for qRel in qrels:
        if qRel["query_num"] in qRelsDict:
            qRelsDict[qRel["query_num"]].append(int(qRel["id"]))
        else:
            qRelsDict[qRel["query_num"]] = [int(qRel["id"])]

    for i in range(0, len(query_ids)):
        totalFScore = totalFScore + self.queryFscore(doc_IDs_ordered[i], query_ids[i], list(qRelsDict[str(query_ids[i])]), k)
    meanFScore =  totalFScore / len(query_ids)
    return meanFScore
```

AP@k:

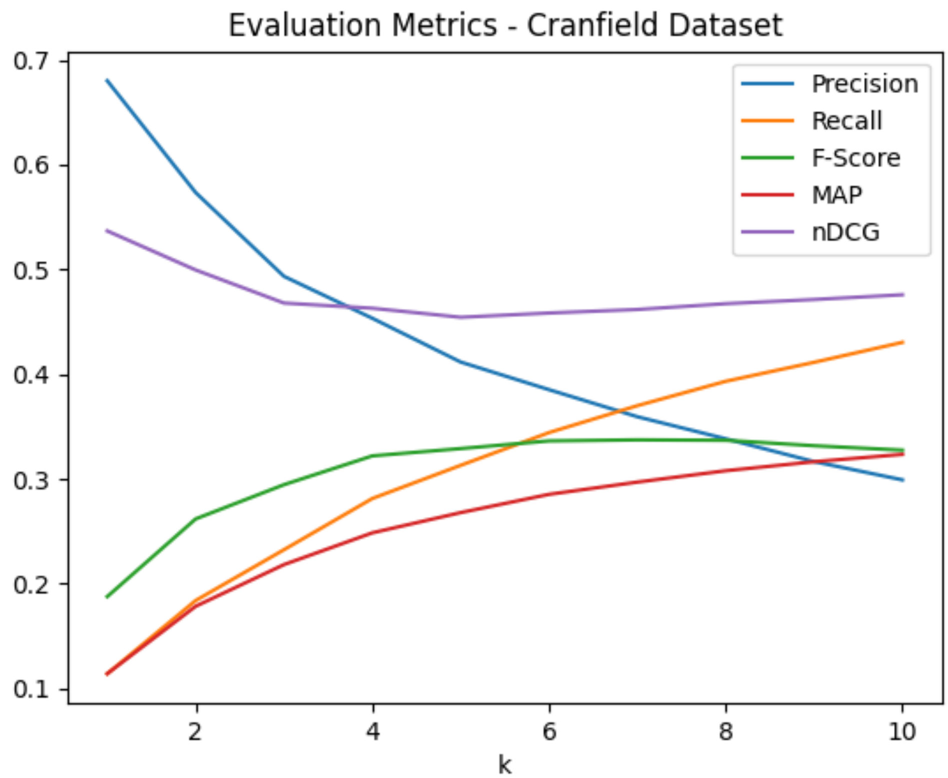
```
    def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        """
        Computation of average precision of the Information Retrieval System
        at a given value of k for a single query (the average of precision@i
        values for i such that the ith document is truly relevant)

        Parameters
        ----------
        arg1 : list
            A list of integers denoting the IDs of documents in
            their predicted order of relevance to a query
        arg2 : int
            The ID of the query in question
        arg3 : list
            The list of documents relevant to the query (ground truth)
        arg4 : int
            The k value

        Returns
        -------
        float
            The average precision value as a number between 0 and 1
        """

        avgPrecision = -1

        #Fill in code here

        relevantCount = 0
        count = 0
        totalPrecision = 0
        for predictedRes in query_doc_IDs_ordered[:k]:
            count = count + 1
            if predictedRes in true_doc_IDs:
                relevantCount = relevantCount + 1
                totalPrecision = totalPrecision + (relevantCount/count)
        avgPrecision = totalPrecision/(len(true_doc_IDs))
        return avgPrecision


    def meanAveragePrecision(self, doc_IDs_ordered, query_ids, q_rels, k):
        """
        Computation of MAP of the Information Retrieval System
        at given value of k, averaged over all the queries

        Parameters
        ----------
        arg1 : list
            A list of lists of integers where the ith sub-list is a list of IDs
            of documents in their predicted order of relevance to the ith query
        arg2 : list
            A list of IDs of the queries
        arg3 : list
            A list of dictionaries containing document-relevance
            judgements - Refer cran_qrels.json for the structure of each
            dictionary
        arg4 : int
            The k value

        Returns
        -------
        float
            The MAP value as a number between 0 and 1
        """

        meanAveragePrecision = -1

        #Fill in code here

        totalAveragePrecision = 0
        qRelsDict = {}
        for qRel in q_rels:
            if qRel["query_num"] in qRelsDict:
                qRelsDict[qRel["query_num"]].append(int(qRel["id"]))
            else:
                qRelsDict[qRel["query_num"]] = [int(qRel["id"])]

        for i in range(0, len(query_ids)):
            totalAveragePrecision = totalAveragePrecision + self.queryAveragePrecision(doc_IDs_ordered[i], query_ids[i],
list(qRelsDict[str(query_ids[i])]), k)
        meanAveragePrecision =  totalAveragePrecision / len(query_ids)
        return meanAveragePrecision
```

nDCG@k:

```python
    def queryNDCG(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        """
        Computation of nDCG of the Information Retrieval System
        at given value of k for a single query

        Parameters
        ----------
        arg1 : list
            A list of integers denoting the IDs of documents in
            their predicted order of relevance to a query
        arg2 : int
            The ID of the query in question
        arg3 : list
            The list of IDs of documents relevant to the query (ground truth)
        arg4 : int
            The k value

        Returns
        -------
        float
            The nDCG value as a number between 0 and 1
        """

        nDCG = -1

        #Fill in code here

        dcg = 0
        count = 1
        for predictedRes in query_doc_IDs_ordered[:k]:
            if predictedRes in true_doc_IDs[0]:
                dcg = dcg + (true_doc_IDs[1][true_doc_IDs[0].index(predictedRes)]/math.log(count+1,2))
            count = count + 1
        idcg = 0
        count = 1
        for idealRes in true_doc_IDs[0][:k]:
            idcg = idcg + (true_doc_IDs[1][count-1]/math.log(count+1,2))
            count = count+1
        nDCG = dcg/idcg

        return nDCG


    def meanNDCG(self, doc_IDs_ordered, query_ids, qrels, k):
        """
        Computation of nDCG of the Information Retrieval System
        at a given value of k, averaged over all the queries

        Parameters
        ----------
        arg1 : list
            A list of lists of integers where the ith sub-list is a list of IDs
            of documents in their predicted order of relevance to the ith query
        arg2 : list
            A list of IDs of the queries for which the documents are ordered
        arg3 : list
            A list of dictionaries containing document-relevance
            judgements - Refer cran_qrels.json for the structure of each
            dictionary
        arg4 : int
            The k value

        Returns
        -------
        float
            The mean nDCG value as a number between 0 and 1
        """

        meanNDCG = -1

        #Fill in code here

        totalNDCG = 0
        qRelsDict = {}
        for qRel in qrels:
            if qRel["query_num"] in qRelsDict:
                qRelsDict[qRel["query_num"]][0].append(int(qRel["id"]))
                qRelsDict[qRel["query_num"]][1].append(5 - int(qRel["position"]))
            else:
                qRelsDict[qRel["query_num"]] = [[int(qRel["id"])],[5 - int(qRel["position"])]]
        for i in range(0, len(query_ids)):
            sortedIndex = np.argsort(list(qRelsDict[str(query_ids[i])])[1])
            docIdList = []
            relList = []
            trueDocIds = []
            for ind in sortedIndex[::-1]:
                docIdList.append(list(qRelsDict[str(query_ids[i])])[0][ind])
                relList.append(list(qRelsDict[str(query_ids[i])])[1][ind])
            trueDocIds.append(docIdList)
            trueDocIds.append(relList)
            totalNDCG = totalNDCG + self.queryNDCG(doc_IDs_ordered[i], query_ids[i], trueDocIds, k)
        meanNDCG = totalNDCG / len(query_ids)
        return meanNDCG
```

2. Assume that for a given query, the set of relevant documents is as listed in incran_qrels.json. Any document with a relevance score of 1 to 4 is considered as relevant. For each query in the Cranfield dataset, find the Precision, Recall, F-score, average precision, and nDCG scores for k = 1 to 10. Average each measure over all queries and plot it as a function of k. The code for plotting is part of the given template. You are expected to use the same. Report the graph with your observations based on it.

Graph:



Observation:

We compared Precision@k and nDCG@k for values k=1 to k=10 to compare our approach with other approaches mentioned in this work. We chose to use Precision and nDCG measures, since it is search engine application and in this case, precision is a bit more important than recall. It is not required for the system to retrieve all relevant documents. Instead it should return the most relevant documents in the first few results. nDCG is another good measure to compare because it also takes into account the graded relevance of the document to the query. Since we have access to the query-document relevance judgements for this dataset, we can leverage these scores to calculate nDCG scores.

3. Using the `time` module in Python, report the run time of your IR system.

Total time taken = 6 minutes 20 seconds

[Warm up] Part 4: Analysis [Theory]

1. What are the limitations of such a Vector space model? Provide examples from the cranfield dataset that illustrate these shortcomings in your IR system.

Limitations:
1. word orders are not taken into account
2. Every coordinate is assumed to be independent of each other.

Examples from your results:

[Refer to the doc: Here]

Part 4: Improving the IR system

Based on the factual record of actual retrieval failures you reported in the assignment, you can develop hypotheses that could address these retrieval failures. You may have to identify the implicit assumptions made by your approach that may have resulted in undesirable results. To realize the improvements, you can use any method(s), including hybrid methods that combine knowledge from linguistic, background, and introspective sources to represent documents. Some examples taught in class are Latent Semantic Analysis (LSA) and Explicit Semantic Analysis (ESA).

You can also explore ways in which a search engine could be improved in aspects such as its efficiency of retrieval, robustness to spelling errors, ability to auto-complete queries, etc.

You are also expected to test these hypotheses rigorously using appropriate hypothesis testing methods. As an outcome of your work, you should be able to make a statement of structure similar to what was presented in the class:

> An algorithm $\boldsymbol{A_1}$ is better than $\boldsymbol{A_2}$ with respect to the evaluation measure $\boldsymbol{E}$ in task $\boldsymbol{T}$ on a specific domain $\boldsymbol{D}$ under certain assumptions $\boldsymbol{A}$.

Note that, unlike the assignment, the scope of this component is open-ended and not restricted to the ideas mentioned here. For each method, the final report must include a critical analysis of results; methods can be combined to come up with improvisations. It is advised that such hybrid methods are well founded on principles and not just ad hoc combinations (an example of an ad hoc approach is a simple convex combination of three methods with parameters tuned to give desired improvements).

You could either build on the template code given earlier for the assignment or develop from scratch as demanded by your approach. Note that while you are free to use any datasets to experiment with, the Cranfield dataset will be used for evaluation. The project will be evaluated based on the rigor in methodology and depth of understanding, in addition to the quality of the report and your performance in Viva.

Your project report (for Part 4) should be well structured and should include the following components.

1. An introduction to the problem setting,
2. The limitations of the basic VSM with appropriate examples from the dataset(s),
3. Your proposed approach(es) to address these issues,
4. A description of the dataset(s) used for experimentations,
5. The results obtained with a comparative study of your approach has improved the IR system, both qualitatively and quantitatively.

The latex template for the final report will be uploaded on Moodle. You are instructed to follow the template strictly.