CS6370: Natural Language Processing
Project

Release Date: 21st March 2024 Deadline:

Name:	Roll No.:
Advait Amit Kisar	AE20B007
Sanket Pramod Bhure	AE20B108
Netheti Devi Varaprasad	ME20B120

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.

To calculate the inverted index representations the data needs to be pre-processed. The code for pre-processing is as follows:-

Importing Necessary Libraries

```
[16] import nltk
     nltk.download('punkt')
     nltk.download('wordnet')
     from nltk.tokenize import word_tokenize
     from nltk.stem import WordNetLemmatizer
     lemmatizer = WordNetLemmatizer()
```

Given three documents

```
[17] d1 = "Herbivores are typically plant eaters and not meat eaters"
     d2 = "Carnivores are typically meat eaters and not plant eaters"
     d3 = "Deers eat grass and leaves"
```

Processing each document

```
[22] stop_words = ["are", "and", "not"] # Assuming {are, and, not} as stop words
     Processed_d1 = [lemmatizer.lemmatize(w) for w in word_tokenize(d1) if w not in stop_words]
     Processed_d2 = [lemmatizer.lemmatize(w) for w in word_tokenize(d2) if w not in stop_words]
     Processed_d3 = [lemmatizer.lemmatize(w) for w in word_tokenize(d3) if w not in stop_words]
```

Processed words given by each document

```
print(Processed_d1)
print(Processed_d2)
print(Processed_d3)
```

```
['Herbivores', 'typically', 'plant', 'eater', 'meat', 'eater']
['Carnivores', 'typically', 'meat', 'eater', 'plant', 'eater']
['Deers', 'eat', 'grass', 'leaf']
```

Source code:
https://colab.research.google.com/drive/1Jut4XhDzyR7Y0Xmvl3F1PeD8pUod_vO7?usp=sharing

Output:

- **d1 = ['Herbivores', 'typically', 'plant', 'eater', 'meat', 'eater']**
- **d2 = ['Carnivores', 'typically', 'meat', 'eater', 'plant', 'eater']**
- **d3 = ['Deers', 'eat', 'grass', 'leaf']**

Given below is the table for term frequencies and inverted index representa--tion for each pre-processed documents.

Words	d1	d2	d3	Inverted Index Representation
Herbivores	1	0	0	[1,0,0]
typically	1	1	0	[1,1,0]
plant	1	1	0	[1,1,0]
eater	2	2	0	[1,1,0]
meat	1	1	0	[1,1,0]
Carnivores	0	1	0	[0,1,0]
Deers	0	0	1	[0,0,1]
eat	0	0	1	[0,0,1]
grass	0	0	1	[0,0,1]
leaf	0	0	1	[0,0,1]

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

Term Frequency (TF) is determined by counting the occurrences of a word (type) within a document. Inverse document frequency (IDF) is calculated here as:

$$\mathbf{IDF = \log_2(N/n)}$$

where N is the total number of documents and n is the number of documents containing a given word (type). The TF-IDF values enable us to depict documents as vectors within a space defined by words (types). In

the expression below, di represents i^{th} document, where j iterates over different types, and k represents the total number of types:

$$di = \sum_{j=1}^{k} tf_{di}(j).idf(j).word(j)$$

TF – IDF Matrix

Words	Inverted Index	tf_{d1}	tf_{d2}	tf_{d3}	idf	d1	d2	d3
Herbivores	[1,0,0]	1	0	0	1.585	1.585	0	0
typically	[1,1,0]	1	1	0	0.585	0.585	0.585	0
plant	[1,1,0]	1	1	0	0.585	0.585	0.585	0
eaters	[1,1,0]	2	2	0	0.585	1.170	1.170	0
meat	[0,1,0]	1	1	0	0.585	0.585	0.585	0
Carnivores	[0,0,1]	0	1	0	1.585	0	1.585	0
Deers	[0,0,1]	0	0	1	1.585	0	0	1.585
eat	[0,0,1]	0	0	1	1.585	0	0	1.585
grass	[0,0,1]	0	0	1	1.585	0	0	1.585
leaves	[0,0,1]	0	0	1	1.585	0	0	1.585

TF-IDF representations of Documents (in word space)

$$d1 = \begin{pmatrix} 1.585 \\ 0.585 \\ 0.585 \\ 1.170 \\ 0.585 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad d2 = \begin{pmatrix} 0 \\ 0.585 \\ 0.585 \\ 1.170 \\ 0.585 \\ 1.585 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad d3 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1.585 \\ 1.585 \\ 1.585 \\ 1.585 \end{pmatrix}$$

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

When utilizing TF-IDF (Term Frequency-Inverse Document Frequency) for document retrieval, documents with non-zero TF-IDF values are retrieved based on the relevance to the query terms. So,

- The documents retrieved from the query '**plant**' will be Document **d1** and Document **d2**.
- The documents retrieved from the query '**eaters**' will be Document **d1** and Document **d2**.

As a result, a query like "**plant eaters**" would retrieve the union of the documents retrieved in the first and second queries. Therefore, Document **d1** and Document **d2** will be retrieved.

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

Cosine Similarity calculations:

Ans: Just as documents are represented as vectors (TF-IDF representation) in information retrieval systems, the query "**plant eaters**" can also be represented in a similar vector format:

$$Q = \begin{bmatrix} 0 \\ 0 \\ 0.585 \\ 0.585 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The **cosine similarity** can be calculated as (say for di):

$$\mathbf{sim(Q, di)} = \frac{\mathbf{Q.di}}{\|\mathbf{Q}\| \, \|\mathbf{di}\|}$$

Magnitude of Document d1:

$\|d1\| = \sqrt{1.585^2 + 0.585^2 + 0.585^2 + 1.170^2 + 0.585^2} = 2.2154$

Magnitude of Document d2:

$\|d2\| = \sqrt{0.585^2 + 0.585^2 + 1.170^2 + 0.585^2 + 1.585^2} = 2.2154$

Magnitude of Document d3:

$\|d3\| = \sqrt{1.585^2 + 1.585^2 + 1.585^2 + 1.585^2 + 1.585^2} = 3.5442$

Magnitude of query Q:

$\|Q\| = \sqrt{0.585^2 + 0.585^2} = 0.8273$

Dot product between d1 and Q:

$\boldsymbol{Q.d1} = 0.585^2 + 0.585^2 = 0.6845$

Dot product between d2 and Q:

$\boldsymbol{Q.d2} = 0.585^2 + 0.585^2 = 0.6845$

Dot product between d3 and Q:

$\boldsymbol{Q.d3} = 0$

Applying the above score (cosine similarity) formula, we find the scores of the three documents with respect to the query:

- Score of Document d1 and query(Q) = **0.3734**
- Score of Document d2 and query(Q) = **0.3734**
- Score of Document d3 and query(Q) = **0**

Ranking documents

Ans: Greater the cosine similarity score, better the rank of the document. The Document **d1** and **d2** have the same cosine similarity for the given query and Document **d3** has the least cosine similarity value. Based on this cosine values we can rank the documents as follows:

Rank 1 → **d1, d2**

Rank 2 → **d3**

Is the ordering desirable? If no, why not?

Ans: No, the current ranking is not desirable. Because, it faces several challenges that impact its effectiveness, key issues are:

- Ambiguity arises when multiple documents receive identical similarity scores. This ambiguity complicates the process of determining which document should be prioritized when they have equal relevance to the query.

- The system's strict reliance on exact word matches leads to the omission of potentially relevant documents, thereby affecting **recall**. For example, document **d3** about "**deers**," "**grass**," and "**leaves**" might be related to a query about "**plant eaters**," but since it doesn't contain the exact words as in the query, it is marked as irrelevant. Hence, it is not retrieved.
- The system may retrieve irrelevant documents (thereby affecting **precision**) based on the presence of query terms, regardless of their contextual relevance. Document **d2**, for example, discusses "**carnivores**," which is unrelated to the query about "**plant eaters**." Nonetheless, it is mistakenly identified as relevant due to the shared presence of query terms.
- Hence, we can observe that the inverted-indexing based retrieval is not a desirable system as it only considers the **lexical relationship** between the documents and queries, which is just one aspect of retrieval. The **semantic relationship** between the two has more importance as compared to **lexical relationship**.

[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.

The code is implemented in a Python file named "**informationRetrieval.py**". Inside this file, there is a class called **InformationRetrieval**. This class contains two functions:

1. **buildIndex:** This function creates a dictionary representing an index of all terms found in a collection of documents. Each term in the dictionary is associated with its frequency in each document and the document IDs in which it appears.

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

    Parameters
    ----------
    docs : 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
    docIDs : list
        A list of integers denoting IDs of the documents

    Returns
    -------
    None
    """
    index = {tokens: [] for d in docs for sentence in d for tokens in sentence}
    for i in range(len(docs)):
        doc = [token for sent in docs[i] for token in sent]
        for j in docs[i]:
            for k in j:
                if k in index.keys():
                    if [docIDs[i], doc.count(k)] not in index[k]:
                        index[k].append([docIDs[i], doc.count(k)])
    self.index = (index, len(docs), docIDs)
```

2. **rank:** This function ranks the documents based on their relevance to a given set of queries. It produces a list of document IDs sorted in decreasing order of ranking.

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

    Parameters
    ----------
    queries : 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 = []
    index, doc_num, doc_ID = self.index

    # Create the term-document matrix D
    D = np.zeros((doc_num, len(index.keys())))
    key = list(index.keys())
    for i in range(len(key)):
        for l in index[key[i]]:
            D[l[0] - 1, i] = l[1]

    # Compute IDF values
    idf = np.zeros((len(key), 1))
    for i in range(len(key)):
        idf[i] = np.log10(doc_num / len(index[key[i]]))

    # Weight the term-document matrix by IDF values
    for i in range(D.shape[0]):
        D[i, :] = D[i, :] * idf.T

    # Process each query
    for i in range(len(queries)):
        query = defaultdict(list)
        for j in queries[i]:
            for k in j:
                if k in index.keys():
                    query[k] = index[k]
        query = dict(query)
        Q = np.zeros((1, len(key)))
        for m in range(len(key)):
            if key[m] in query.keys():
                Q[0, m] = 1
        Q = Q * idf.T

        # Compute cosine similarity between query and documents
        similarities = []
        for d in range(D.shape[0]):
            norm_query = np.linalg.norm(Q[0, :]) + 1e-4
            norm_doc = np.linalg.norm(D[d, :]) + 1e-4
            simi = np.dot(Q[0, :], D[d, :])/(norm_query*norm_doc)
            similarities.append(simi)

        # Sort documents based on similarity scores
        doc_IDs_ordered.append([x for _, x in sorted(zip(similarities, doc_ID), reverse=True)])
    return doc_IDs_ordered
```

Output Samples:

- Query: "**papers on aerodynamics**"
  Top five document IDs = [46, 875, 388, 634, 739]
- Query: "**what chemical kinetic system is applicable to hypersonic aerodynamic problems**"
  Top five document IDs = [103, 943, 746, 410, 1032]
- Query: "**papers on internal /slip flow/ heat transfer studies**"
  Top five document IDs = [102, 45, 46, 846, 763]

[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:

Computation of **precision** of the Information Retrieval System at a given value of k for a single query:

```
def query_precision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    # Compute the total number of documents retrieved
    num_docs = len(query_doc_IDs_ordered)
    # Check if the value of k is valid
    if k > num_docs:
        print("Insufficient documents retrieved for given k")
        return -1
    # Initialize a variable to count the number of relevant documents
    num_relevant_docs = 0
    # Iterate over the top k ranked documents
    for doc_id in query_doc_IDs_ordered[:k]:
        # Check if the document ID is among the relevant documents
        if int(doc_id) in true_doc_IDs:
            # If the document is relevant, increment the count
            num_relevant_docs += 1
    # Compute and return the precision value
    precision = num_relevant_docs / k
    return precision
```

Computation of **mean precision** of the Information Retrieval System at a given value of k, averaged over all the queries:

```
def meanPrecision(self, doc_ids_ordered, query_ids, qrels, k):
    num_queries = len(query_ids)
    precisions = []
    # Check if the number of queries and documents match
    if len(doc_ids_ordered) != num_queries:
        print("Number of queries and documents do not match")
        return -1
    # Iterate over all queries
    for i in range(num_queries):
        query_docs = doc_ids_ordered[i]
        query_id = int(query_ids[i])
        # Retrieve relevant documents for the query
        relevant_docs = [int(entry["id"]) for entry in qrels if int(entry["query_num"]) == query_id]
        # Compute precision for the query
        precision = self.query_precision(query_docs, query_id, relevant_docs, k)
        precisions.append(precision)
    # Compute and return the mean precision value
    if precisions:
        return sum(precisions) / num_queries
    else:
        print("Empty list.")
        return -1
```

Recall@k:

Computation of **recall** of the Information Retrieval System at a given value of k for a single query:

```
def query_recall(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    num_docs = len(query_doc_IDs_ordered)
    num_relevant_docs = len(true_doc_IDs)
    # Check if the number of documents retrieved is sufficient for k
    if k > num_docs:
        print("Insufficient number of retrieved documents for given k")
        return -1
    # Count the number of relevant documents retrieved within top k
    num_retrieved_relevant_docs = 0
    # Iterate over the top k ranked documents
    for doc_id in query_doc_IDs_ordered[:k]:
        # Check if the document ID is among the relevant documents
        if int(doc_id) in true_doc_IDs:
            # If the document is relevant, increment the count
            num_retrieved_relevant_docs += 1
    # Compute and return recall value
    return num_retrieved_relevant_docs / num_relevant_docs if num_relevant_docs != 0 else 0
```

Computation of **mean recall** of the Information Retrieval System at a given value of k, averaged over all the queries:

```python
def meanRecall(self, doc_ids_ordered, query_ids, qrels, k):
    num_queries = len(query_ids)
    recalls = []
    # Check if the number of queries and documents match
    if len(doc_ids_ordered) != num_queries:
        print("Number of queries and documents do not match")
        return -1
    # Iterate over all queries
    for i in range(num_queries):
        query_docs = doc_ids_ordered[i]
        query_id = query_ids[i]
        # Retrieve relevant documents for the query
        relevant_docs = [int(entry["id"]) for entry in qrels if int(entry["query_num"]) == query_id]
        # Compute recall for the query
        recall = self.query_recall(query_docs, query_id, relevant_docs, k)
        recalls.append(recall)
    # Compute and return the mean recall value
    if recalls:
        return sum(recalls) / num_queries
    else:
        print("List is empty.")
        return -1
```

$F_{0.5}$ score@k:

Computation of **f-score** of the Information Retrieval System at a given value of k for a single query:

```python
def query_fscore(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    # Compute precision and recall for the query
    precision = self.query_precision(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    recall = self.query_recall(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    # Compute fscore based on precision and recall
    if precision > 0 and recall > 0:
        fscore = 2 * precision * recall / (precision + recall)
    else:
        fscore = 0
    return fscore
```

Computation of **mean f-score** of the Information Retrieval System at a given value of k, averaged over all the queries:

```python
def meanFscore(self, doc_ids_ordered, query_ids, qrels, k):
    num_queries = len(query_ids)
    fscores = []
    # Check if the number of queries and documents match
    if len(doc_ids_ordered) != num_queries:
        print("Number of queries and documents do not match")
        return -1
    # Iterate over all queries
    for i in range(num_queries):
        query_docs = doc_ids_ordered[i]
        query_id = query_ids[i]
        # Retrieve relevant documents for the query
        relevant_docs = [int(entry["id"]) for entry in qrels if int(entry["query_num"]) == query_id]
        # Compute fscore for the query
        fscore = self.query_fscore(query_docs, query_id, relevant_docs, k)
        fscores.append(fscore)
    # Compute and return the mean fscore value
    if fscores:
        return sum(fscores) / num_queries
    else:
        print("Empty list")
        return -1
```

AP@k:

Computation of **average precision** of the Information Retrieval System at a given value of k for a single query:

```python
def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    num_true_docs = len(true_doc_IDs)
    num_docs_retrieved = len(query_doc_IDs_ordered)
    # Check if enough documents are retrieved for the given k
    if k > num_docs_retrieved:
        print("Insufficient documents retrieved for given k")
        return -1
    relevances = [1 if int(doc_ID) in true_doc_IDs else 0 for doc_ID in query_doc_IDs_ordered]
    # Calculate precision@i for each document up to k
    precisions = [self.query_precision(query_doc_IDs_ordered, query_id, true_doc_IDs, i) for i in range(1, k + 1)]
    # Calculate precision at k multiplied by relevance for each document
    precision_at_k = [precisions[i] * relevances[i] for i in range(k)]
    # Calculate average precision
    if num_true_docs != 0:
        if sum(relevances[:k]) != 0:
            AveP = sum(precision_at_k) / len(true_doc_IDs)
        else:
            AveP = 0
        return AveP
    else:
        print("No true documents are present.")
        return -1
```

Computation of **mean average precision** of the Information Retrieval System at a given value of k, averaged over all the queries:

```python
def meanAveragePrecision(self, doc_IDs_ordered, query_ids, q_rels, k):
    num_queries = len(query_ids)
    map = []
    # Check if the number of queries and documents match
    if len(doc_IDs_ordered) != num_queries:
        print("Number of queries and documents do not match")
        return -1
    # Iterate over all queries
    for i in range(num_queries):
        query_docs = doc_IDs_ordered[i]
        query_id = query_ids[i]
        # Retrieve relevant documents for the query
        relevant_docs = [int(entry["id"]) for entry in q_rels if int(entry["query_num"]) == query_id]
        # Compute average precision for the query
        ap = self.queryAveragePrecision(query_docs, query_id, relevant_docs, k)
        map.append(ap)
    # Compute and return the mean average precision value
    if map:
        return sum(map) / num_queries
    else:
        print("Empty list")
        return -1
```

nDCG@k:

Computation of **nDCG** of the Information Retrieval System at a given value of k for a single query:

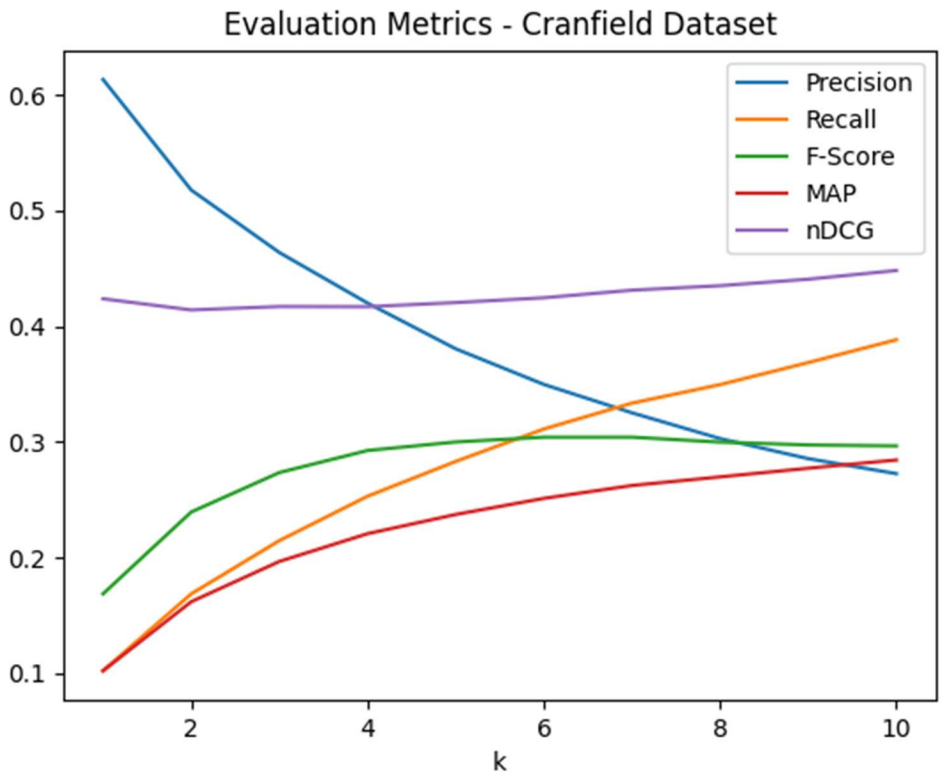
```python
def query_nDCG(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    # Create a dictionary to store relevant documents for the given query
    relevant_docs = {}
    for doc in true_doc_IDs:
        if int(doc["query_num"]) == int(query_id):
            doc_id = int(doc["id"])
            relevance = 5 - doc["position"]  # Calculate relevance score
            relevant_docs[doc_id] = relevance
    # Calculate DCG (Discounted Cumulative Gain) at position k
    DCG_k = 0
    for rank in range(1, min(k, len(query_doc_IDs_ordered)) + 1):
        doc_ID = int(query_doc_IDs_ordered[rank - 1])
        if doc_ID in relevant_docs:
            relevance = relevant_docs[doc_ID]
            DCG_k += (2 ** relevance - 1) / log2(rank + 1)  # Update DCG_k
    # Calculate IDCG (Ideal Discounted Cumulative Gain) at position k
    sorted_relevances = sorted(relevant_docs.values(), reverse=True)
    IDCG_k = sum((2 ** relevance - 1) / log2(rank + 1) for rank, relevance in enumerate(sorted_relevances, 1) if rank <= k)
    # Check if IDCG_k is not zero, then compute nDCG
    if IDCG_k != 0:
        nDCG_k = DCG_k / IDCG_k
        return nDCG_k
    else:
        # If IDCG_k is zero, print a message and return -1
        print("IDCG_k is zero.")
        return -1
```

Computation of **mean nDCG** of the Information Retrieval System at a given value of k, averaged over all the queries:

```python
def meanNDCG(self, doc_IDs_ordered, query_ids, qrels, k):
    num_queries = len(query_ids)
    ndcgs = []
    # Check if the number of queries and documents match
    if len(doc_IDs_ordered) != num_queries:
        print("Number of queries and documents do not match")
        return -1
    # Iterate over all queries
    for i in range(num_queries):
        query_doc = doc_IDs_ordered[i]
        query_id = int(query_ids[i])
        nDCG = self.query_nDCG(query_doc, query_id, qrels, k)
        ndcgs.append(nDCG)
    # Compute and return the mean nDCG value
    if ndcgs:
        return sum(ndcgs) / num_queries
    else:
        print("Empty list")
        return -1
```

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:

1) **Precision:** As we retrieve more documents (increasing k), precision tends to **decrease**. Initially, the top-ranked documents are highly relevant, resulting in high precision. However, as k increases, the search engine may include less relevant or irrelevant documents in the retrieved set, leading to a decrease in the proportion of relevant documents and hence lowering precision**.**
2) **Recall:** As k increases, recall either **increases** or stays **constant**. This is because the total number of relevant documents in the corpus remains constant (denominator), while the number of relevant documents retrieved can only increase or remain the same (numerator) as more documents are retrieved. Therefore, recall tends to increase or stay constant with increasing k.
3) **F-score:** The F-score, which balances Precision and Recall, reaches **saturation** after **k ≥ 6**. In scenarios where equal importance is given to both Precision and Recall, the model's performance stabilizes after **k ≥ 6**.
4) **MAP:** As k **increases**, MAP tends to **increase**. This is because the

average precision calculation, which is a key component of MAP, involves summing precision values across all relevant documents and dividing by the total number of relevant documents, which remains **constant**. Therefore, considering more documents generally leads to a higher average precision and thus an increase in MAP.

5) **nDCG:** It shows stability across different k values, reflecting consistent IR system performance. This indicates effective document ranking based on **relevance**, irrespective of specific k values.

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

Execution time: 13.698672533035278 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) The vector space model uses **high-dimensional** vectors, making it **computationally** intensive.
2) The vector space model may produce **low** similarity scores for similar documents with different vocabularies, as **synonyms** are treated as distinct words, potentially leading to **false negatives** and reduced **recall** and **failure** to retrieve relevant documents.
3) The vector space model doesn't consider word meaning in context, causing queries with **polysemous words** to match irrelevant documents, reducing **precision**.
4) The **order** in which the terms appear in the document is lost in the vector space representation.
5) The model implicitly assumes the terms are **orthogonal**, which is often not the case.
6) The changes in **case** of the query creates changes in the retrieved documents.

Examples from your results:

1) **Same Context:** Both the below queries have the same context of high velocity low pressure condition, but the sets of retrieved documents are different.
   a. Query: **"they fly at high velocity"**
      Top 5 Document IDs: [933, 1147, 687, 378, 774]
   b. Query: **"they fly at low pressure"**
      Top 5 Document IDs: [238, 933, 10, 919, 1147]

2) **Synonyms:** The two pairs of synonyms {aircrafts, jets} and {speeds, mach. no} are used here. However, the retrieved documents are different.
   a. Query: **"aircrafts fly at high speeds"**
      Top 5 Document IDs: [316, 1147, 933, 41, 214]
   b. Query: **"supersonic jets fly at high mach no."**
      Top 5 Document IDs: [997, 1223, 696, 177, 86]

3) **Polysemous words:** The polysemous word used is "stall" where, in aerodynamics, it refers to the sudden decrease in lift force causing instability. Whereas, in the 2nd query, it refers to a shop where books are available. We expect the documents to be different, but the retrieved documents are more or less the same.
    a. Query: **"The aeroplane is undergoing stall"**
       Top 5 Document IDs: [589, 441, 783, 444, 588]
    b. Query: **"The books are available at the stall"**
       Top 5 Document IDs: [589, 441, 1174, 444, 588]

4) **Case Dependency:** The retrieval is dependent on case of the letters of the query. For different cases for the same letter(s), the retrieved documents are different. This is not expected.
    a. Query: **"Aerodynamics is a field in which the effect of wind on airplane because of the Lift and Drag forces is studied"**
       Top 5 Document IDs: [783, 267, 61, 1153, 673]
    b. Query: **"aerodynamics is a field in which the effect of wind on airplane because of the lift and drag forces is studied"**
       Top 5 Document IDs: [1380, 360, 783, 279, 1256]

Summary:

It can be observed that the IR system is making errors using the above examples. This is because of the fact that it only looks at the **lexical relationship** of the documents and queries instead of the **semantic relationships** which are much more important in case of retrieval of documents.

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.