CS6370: Natural Language Processing
Project

Release Date: 24th March 2024 Deadline: 12th May 2025

Name: Roll No.:

ASHWIN U	CH22B025
ARYAN AGRAWAL	CH22B056
ROOPESH P	CH22B093
SHUHAIBALI VN	EP20B037
AMISHI PANDE	MM23B037

General Instructions:

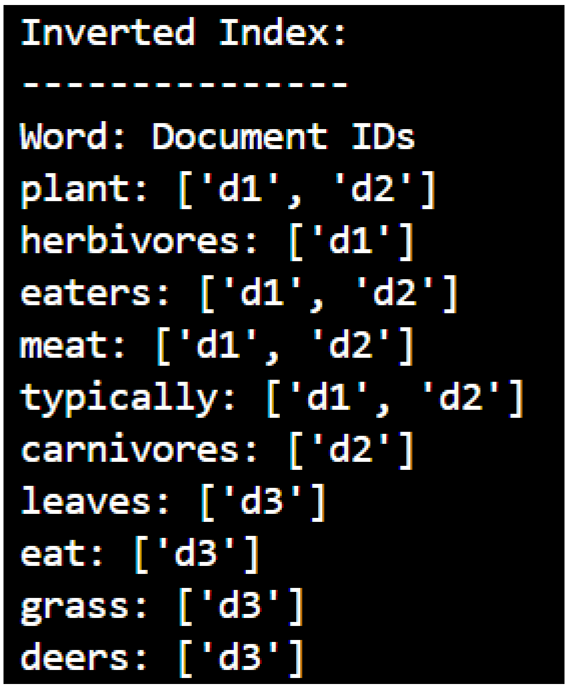
1. The template for the code (in Python) is provided in a separate zip file. You are expected to fill in the template wherever instructed. Note that any Python library, such as nltk, stanfordcorenlp, spacy, etc, can be used.
2. A folder named 'Roll_number.zip' that contains a zip of the code folder and 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 system
[Numerical]
Consider the following three documents:

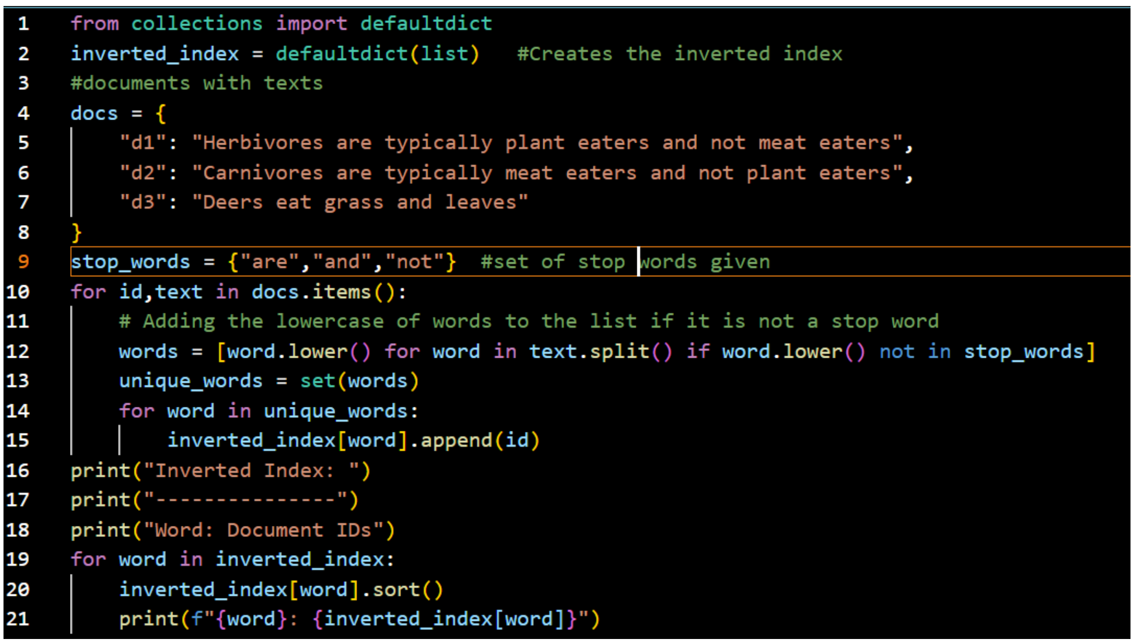
d_1: Herbivores are typically plant eaters and not meat eaters
d_2: Carnivores are typically meat eaters and not plant eaters
d_3: Deers eat grass and leaves

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



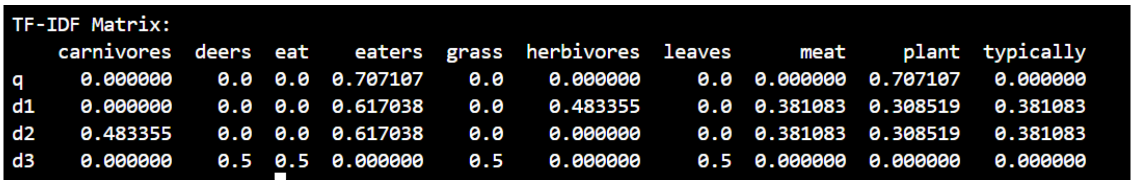
```
Inverted Index:
---------------
Word: Document IDs
plant: ['d1', 'd2']
herbivores: ['d1']
eaters: ['d1', 'd2']
meat: ['d1', 'd2']
typically: ['d1', 'd2']
carnivores: ['d2']
leaves: ['d3']
eat: ['d3']
grass: ['d3']
deers: ['d3']
```

Code:



```python
from collections import defaultdict
inverted_index = defaultdict(list)   #Creates the inverted index
#documents with texts
docs = {
    "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"
}
stop_words = {"are","and","not"}  #set of stop words given
for id,text in docs.items():
    # Adding the lowercase of words to the list if it is not a stop word
    words = [word.lower() for word in text.split() if word.lower() not in stop_words]
    unique_words = set(words)
    for word in unique_words:
        inverted_index[word].append(id)
print("Inverted Index: ")
print("---------------")
print("Word: Document IDs")
for word in inverted_index:
    inverted_index[word].sort()
    print(f"{word}: {inverted_index[word]}")
```

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

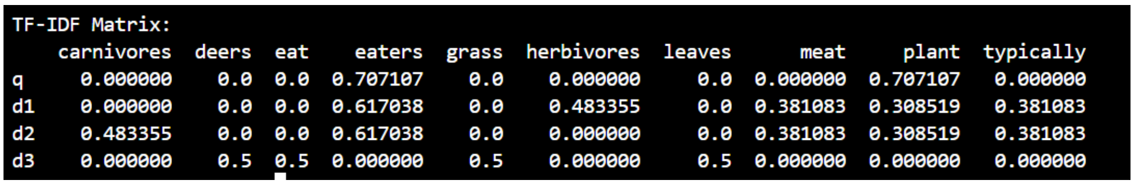


```
TF-IDF Matrix:
    carnivores  deers  eat    eaters  grass  herbivores  leaves      meat     plant  typically
q     0.000000    0.0  0.0  0.707107    0.0    0.000000     0.0  0.000000  0.707107   0.000000
d1    0.000000    0.0  0.0  0.617038    0.0    0.483355     0.0  0.381083  0.308519   0.381083
d2    0.483355    0.0  0.0  0.617038    0.0    0.000000     0.0  0.381083  0.308519   0.381083
d3    0.000000    0.5  0.5  0.000000    0.5    0.000000     0.5  0.000000  0.000000   0.000000
```

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

Based on the inverted index constructed before, the documents that would be retrieved for the given query are **d1 and d2**.

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

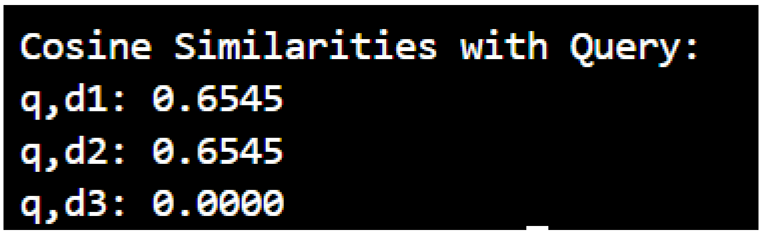


```
TF-IDF Matrix:
    carnivores  deers  eat    eaters  grass  herbivores  leaves      meat     plant  typically
q     0.000000    0.0  0.0  0.707107    0.0    0.000000     0.0  0.000000  0.707107   0.000000
d1    0.000000    0.0  0.0  0.617038    0.0    0.483355     0.0  0.381083  0.308519   0.381083
d2    0.483355    0.0  0.0  0.617038    0.0    0.000000     0.0  0.381083  0.308519   0.381083
d3    0.000000    0.5  0.5  0.000000    0.5    0.000000     0.5  0.000000  0.000000   0.000000
```

Cosine Similarity calculations: $|q| = ((0.707107)^2 + (0.707107)^2)^{0.5} = 1$, $|d1| = ((0.617038)^2 + (0.483355)^2 + (0.381083)^2 + (0.308519)^2 +$

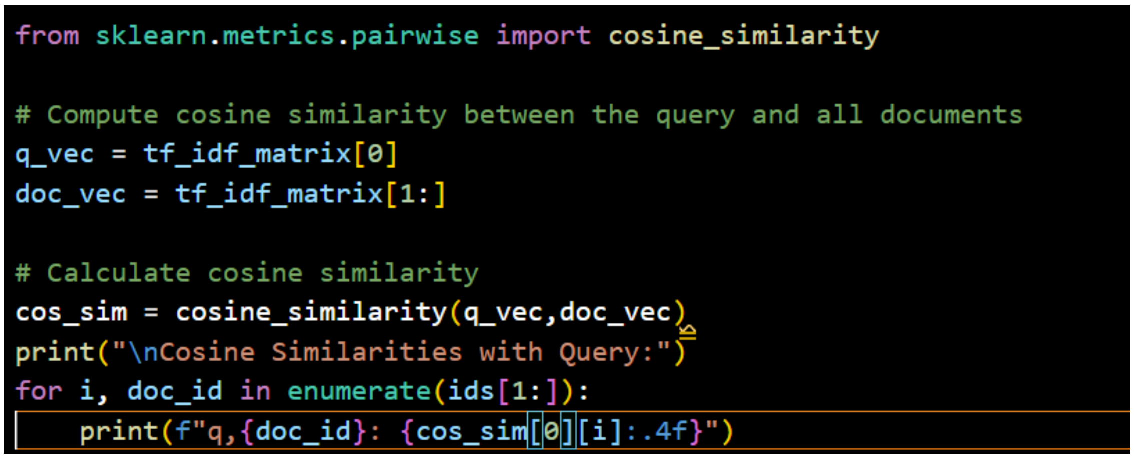
$(0.381083)^2)^{0.5} = 1$, $|d2| = ((0.483355)^2 + (0.617038)^2 + (0.381083)^2 + (0.308519)^2 + (0.381083)^2)^{0.5} = 1$, $|d3| = ((0.5)^2 + (0.5)^2 + (0.5)^2 + (0.5)^2)^{0.5} = 1$

Sim(q,d1) = q.d1/(|q|*|d1|) = 0.6545/(1*1) = 0.6545
Sim(q,d2) = q.d2/(|q|*|d2|) = 0.6545/(1*1) = 0.6545
Sim(q,d3) = q.d3/(|q|*|d3|) = 0



```
Cosine Similarities with Query:
q,d1: 0.6545
q,d2: 0.6545
q,d3: 0.0000
```

Code:



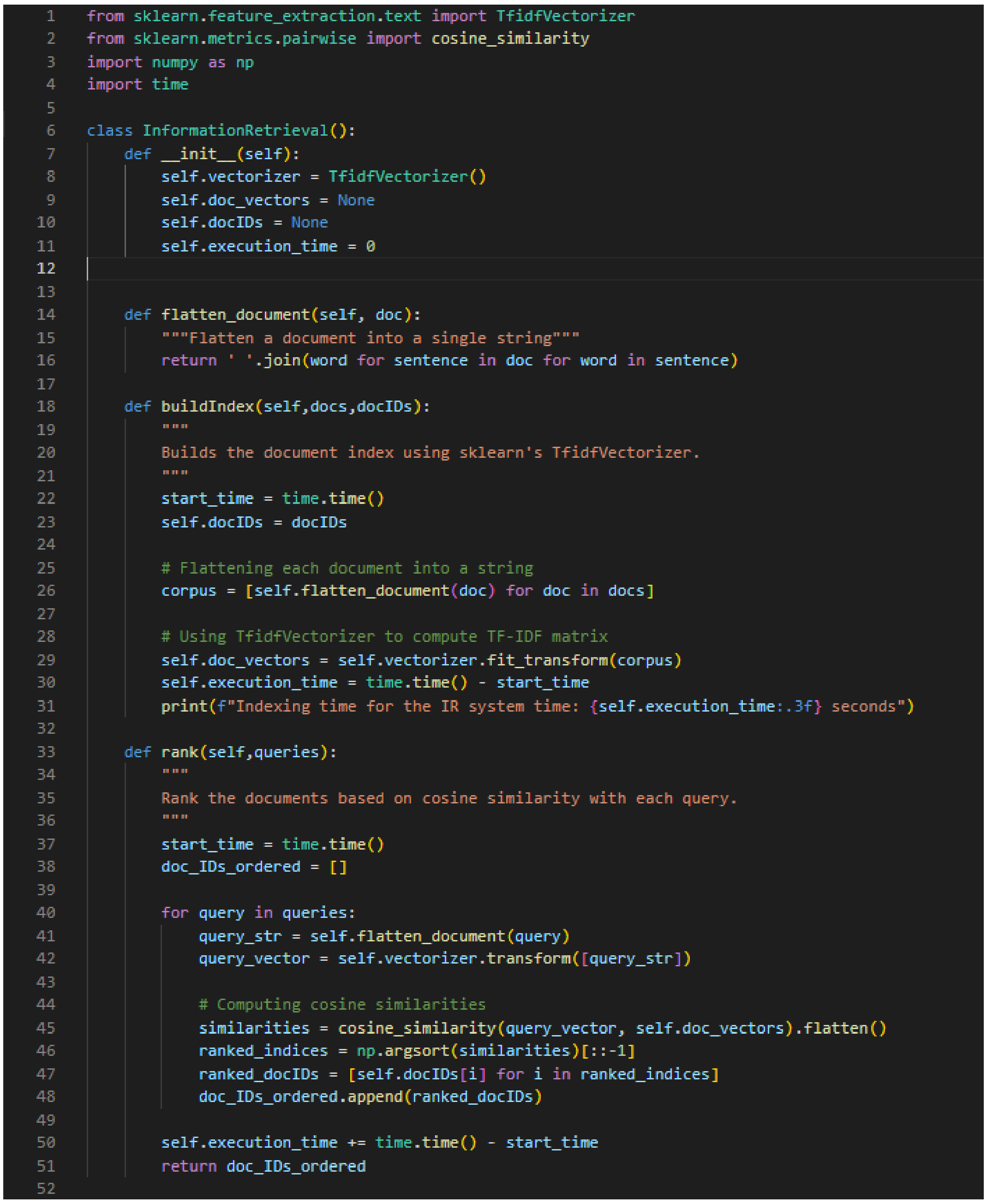
```python
from sklearn.metrics.pairwise import cosine_similarity

# Compute cosine similarity between the query and all documents
q_vec = tf_idf_matrix[0]
doc_vec = tf_idf_matrix[1:]

# Calculate cosine similarity
cos_sim = cosine_similarity(q_vec,doc_vec)
print("\nCosine Similarities with Query:")
for i, doc_id in enumerate(ids[1:]):
    print(f"q,{doc_id}: {cos_sim[0][i]:.4f}")
```

Ranking documents:
Based on the cosine similarity calculations, the documents rank are: d2=d1>d3.

Is the ordering desirable? If no, why not?:
The ordering is **not desirable.** This is because, by observation, it is clear that the query is **semantically** related to the documents d1 and d3 (which talks about herbivores and deers, that eat plants). However, we get **d2** to be the document which has the **highest cosine similarity** which has no association to "plant eaters" and thus is irrelevant and **d3 has the least**.

[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
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

class InformationRetrieval():
    def __init__(self):
        self.vectorizer = TfidfVectorizer()
        self.doc_vectors = None
        self.docIDs = None
        self.execution_time = 0


    def flatten_document(self, doc):
        """Flatten a document into a single string"""
        return ' '.join(word for sentence in doc for word in sentence)

    def buildIndex(self,docs,docIDs):
        """
        Builds the document index using sklearn's TfidfVectorizer.
        """
        start_time = time.time()
        self.docIDs = docIDs

        # Flattening each document into a string
        corpus = [self.flatten_document(doc) for doc in docs]

        # Using TfidfVectorizer to compute TF-IDF matrix
        self.doc_vectors = self.vectorizer.fit_transform(corpus)
        self.execution_time = time.time() - start_time
        print(f"Indexing time for the IR system time: {self.execution_time:.3f} seconds")

    def rank(self,queries):
        """
        Rank the documents based on cosine similarity with each query.
        """
        start_time = time.time()
        doc_IDs_ordered = []

        for query in queries:
            query_str = self.flatten_document(query)
            query_vector = self.vectorizer.transform([query_str])

            # Computing cosine similarities
            similarities = cosine_similarity(query_vector, self.doc_vectors).flatten()
            ranked_indices = np.argsort(similarities)[::-1]
            ranked_docIDs = [self.docIDs[i] for i in ranked_indices]
            doc_IDs_ordered.append(ranked_docIDs)

        self.execution_time += time.time() - start_time
        return doc_IDs_ordered

```

[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):
        if k <= 0 or not query_doc_IDs_ordered:
            return 0.0
        top_k_docs = query_doc_IDs_ordered[:k]
     relevant_count = sum(1 for doc_id in top_k_docs if doc_id in true_doc_IDs)
        precision = relevant_count/min(k,len(query_doc_IDs_ordered))
        return precision
```

Recall@k:

```
def queryRecall(self,query_doc_IDs_ordered,query_id,true_doc_IDs,k):
        if not true_doc_IDs or k <= 0 or not query_doc_IDs_ordered:
            return 0.0
        top_k_docs = query_doc_IDs_ordered[:k]
     relevant_count = sum(1 for doc_id in top_k_docs if doc_id in true_doc_IDs)
        recall = relevant_count/len(true_doc_IDs)
        return recall
```

$F_{0.5}$ score@k:

```
def queryFscore(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        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 + recall == 0:
            return 0.0
        fscore = 2*precision*recall/(precision + recall)
        return fscore
```

AP@k:

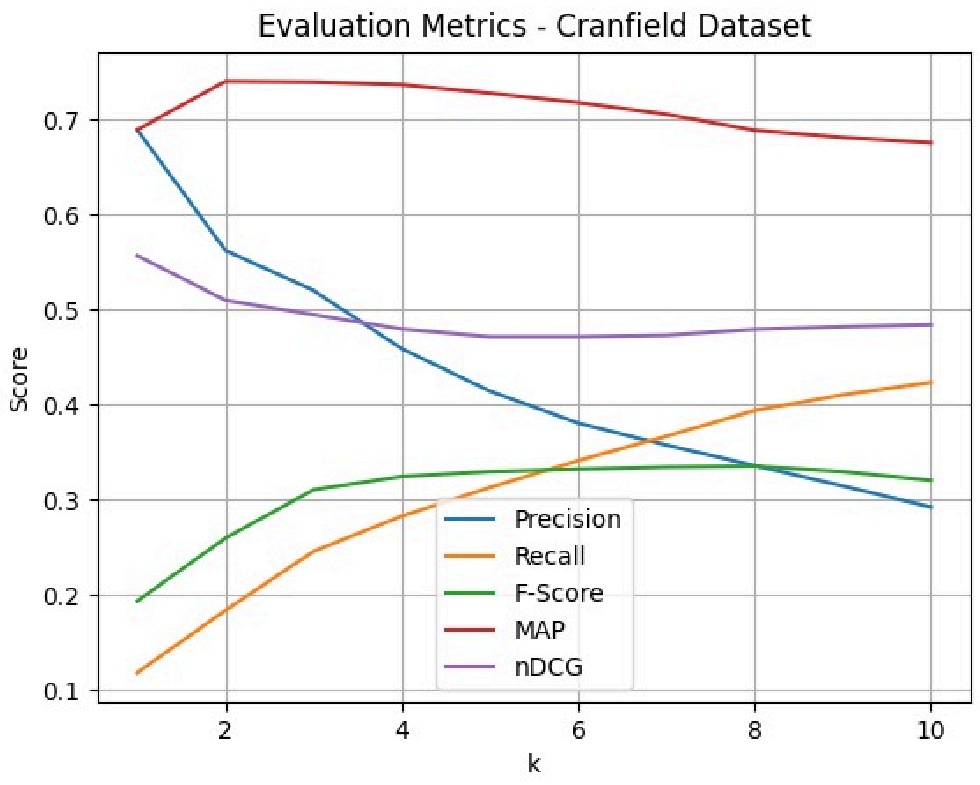
```
def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        if not query_doc_IDs_ordered or not true_doc_IDs or k <= 0:
            return 0.0
        top_k_docs = query_doc_IDs_ordered[:k]
        num_relevant = 0
        precision_sum = 0.0
        for i, doc_id in enumerate(top_k_docs):
            if doc_id in true_doc_IDs:
                num_relevant += 1
                precision_at_i = num_relevant/(i + 1)
                precision_sum += precision_at_i
        if num_relevant == 0:
            return 0.0
        avg_precision = precision_sum/num_relevant
        return avg_precision
```

nDCG@k:

```
def queryNDCG(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        if not query_doc_IDs_ordered or not true_doc_IDs or k <= 0:
            return 0.0
        top_k_docs = query_doc_IDs_ordered[:k]
        dcg = 0.0
        for i,doc_id in enumerate(top_k_docs):
            rel = true_doc_IDs.get(doc_id,0)
            dcg += rel/math.log2(i + 2)
        ideal_rels = sorted(true_doc_IDs.values(),reverse = True)[:k]
        idcg =
sum(rel/math.log2(i + 2) for i,rel in enumerate(ideal_rels))
        if idcg == 0:
            return 0.0
        nDCG = dcg/idcg
        return nDCG
```

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 decreases** as k increases:
    a. Highest at k=1 (0.6889) which drops to 0.2924 at k=10.
    b. This is expected because top-ranked documents are usually the most relevant, and adding more results introduces less relevant ones.
2. **Recall increases** with k:
    a. Starts from 0.1183 at k=1, improves steadily to 0.4235 at k=10.
    b. This is also expected as more retrieved documents means more chances to include relevant ones.
3. **F-score** (harmonic mean of precision and recall) improves initially but then stabilizes indicating a balance point between precision and recall.
4. **MAP (Mean Average Precision)** slightly decreases with k:
    a. Starts high at 0.7400 around k=2, ends at 0.6754 at k=10.
    b. Reflects that early precision in ranking is strong but decreases with k.
5. **nDCG** (Normalized Discounted Cumulative Gain) remains relatively stable which indicates that the system does a decent job of

placing relevant documents early in the ranking.

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

0.858

[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:

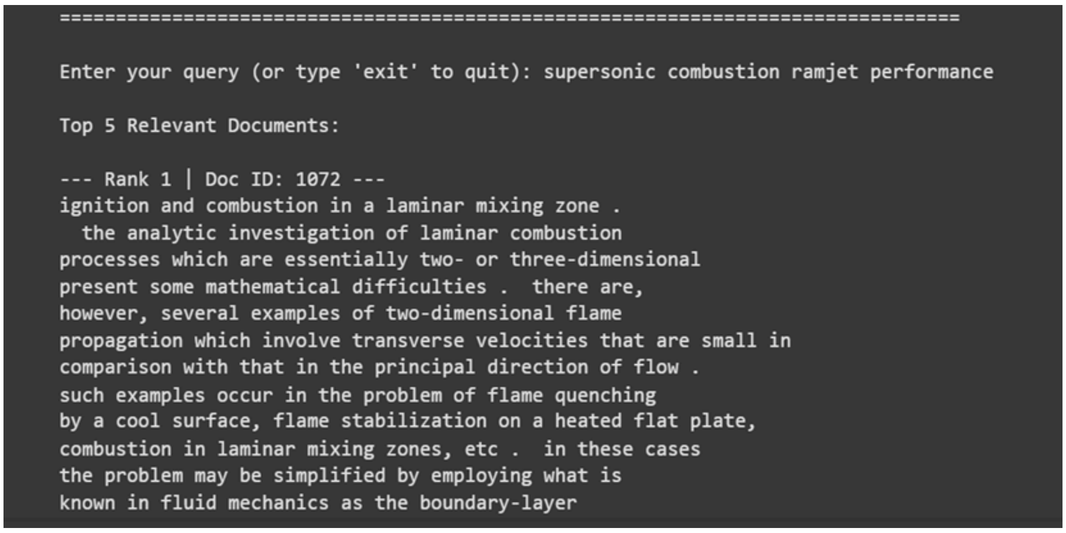
1) Lack of semantic Understanding:
   - Different words with same meaning are treated as entirely different in the vector space.
   - Model can't distinguish same word in different context.

2) VSM assumes term independence, ignoring syntactic and semantic relationships between words:
   - Model struggles with phrases where word order matters.
   - Relationships between words in a sentence is completely lost. Which can lead to poor retrieval performance for queries where context is important.

3) High dimensionality issues:
   - As the vocabulary size increases, dimensionality of vector space grows, making similarity calculations less effective.
   - Most document vectors contains many zeros, which can affect similarity calculations, thus retrieval performance.

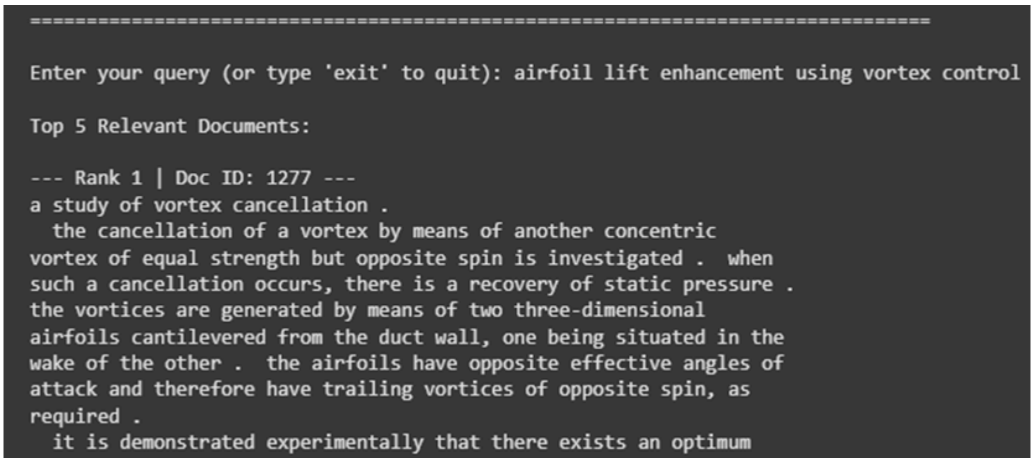
Examples from your results:

1) For the query: "supersonic combustion ramjet performance": the IR system retrieved doc – 1072("ignition and combustion in a laminar mixing zone .") at rank 1. Which proves that as the keywords in the query keeps increasing model dilutes the meaning as a result the one with more keywords are chosen.



```
================================================================================

Enter your query (or type 'exit' to quit): supersonic combustion ramjet performance

Top 5 Relevant Documents:

--- Rank 1 | Doc ID: 1072 ---
ignition and combustion in a laminar mixing zone .
  the analytic investigation of laminar combustion
processes which are essentially two- or three-dimensional
present some mathematical difficulties .  there are,
however, several examples of two-dimensional flame
propagation which involve transverse velocities that are small in
comparison with that in the principal direction of flow .
such examples occur in the problem of flame quenching
by a cool surface, flame stabilization on a heated flat plate,
combustion in laminar mixing zones, etc .  in these cases
the problem may be simplified by employing what is
known in fluid mechanics as the boundary-layer
```

2) The TF-IDF-based Vector Space Model is its inability to capture semantic meaning. For instance, when querying “airfoil lift enhancement using vortex control,” the top-ranked document (Doc ID 1277) focuses on vortex cancellation and pressure recovery, not on enhancing lift using vortex-based techniques. While it shares surface-level terms like “vortex” and “airfoil,” it fails to understand the actual intent behind the query, showing how TF-IDF can retrieve topically related but semantically irrelevant documents.



```
================================================================================

Enter your query (or type 'exit' to quit): airfoil lift enhancement using vortex control

Top 5 Relevant Documents:

--- Rank 1 | Doc ID: 1277 ---
a study of vortex cancellation .
  the cancellation of a vortex by means of another concentric
vortex of equal strength but opposite spin is investigated .  when
such a cancellation occurs, there is a recovery of static pressure .
the vortices are generated by means of two three-dimensional
airfoils cantilevered from the duct wall, one being situated in the
wake of the other .  the airfoils have opposite effective angles of
attack and therefore have trailing vortices of opposite spin, as
required .
  it is demonstrated experimentally that there exists an optimum
```