# CS6370: Natural Language Processing
Project

Release Date: 24th March 2024 Deadline: 8th May 2025

| Name: | Roll No.: |
|---|---|
| NAGRAJ MANJUNATH GAONKAR | CE23B048 |
| VED NITIN KORDE | CE23B123 |
| MOHAMED ZAYAAN S | CE23B092 |
| KAMESH P | ED20B027 |
| MIDDE GNANA ABHISHEK | CE21B053 |

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.

---

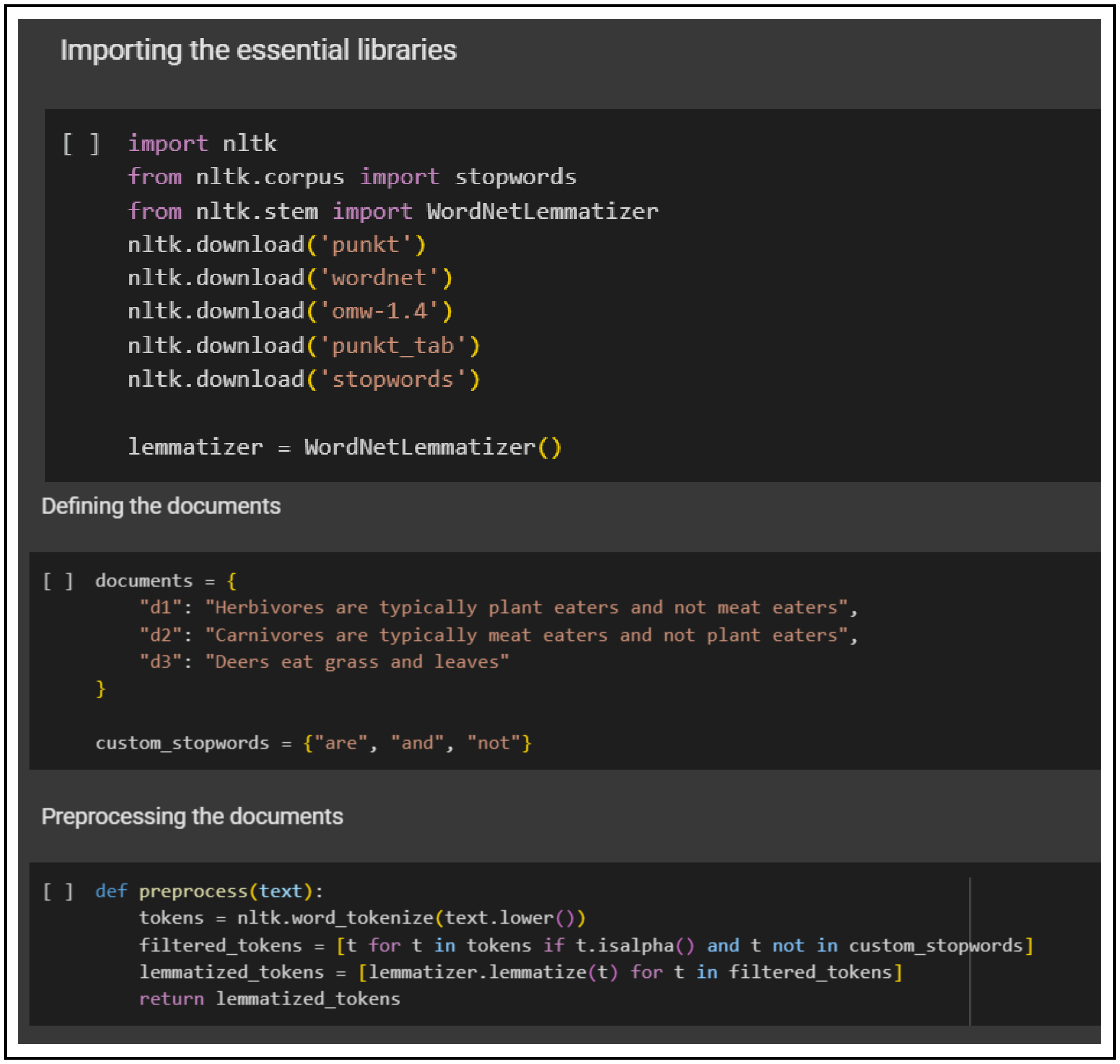
The first assignment in the NLP course involved building a basic text processing module that implements sentence segmentation, tokenization, stemming /lemmatization, stopword removal, and some aspects of spell check. This module involves implementing an Information Retrieval system using the Vector Space 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.



Importing the essential libraries

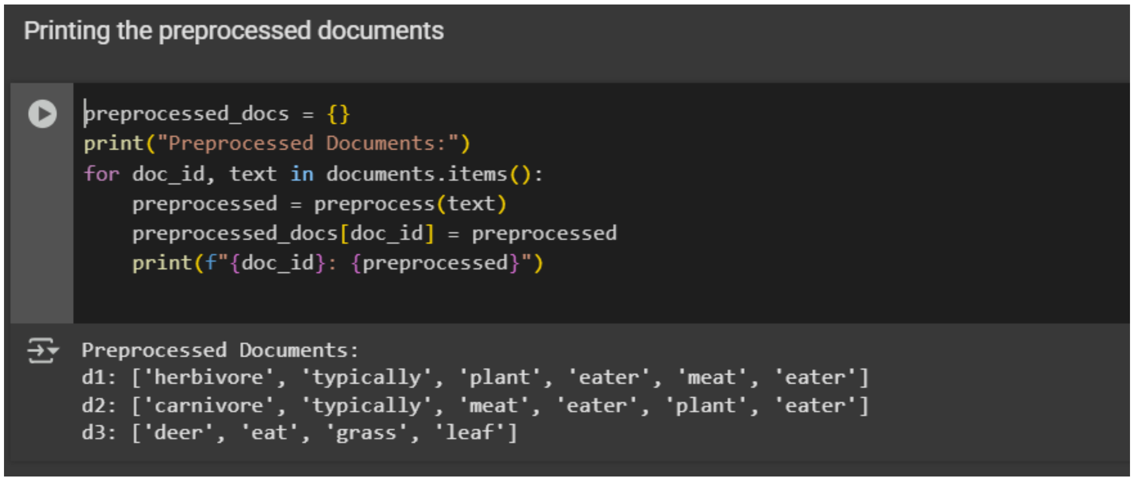
```
[ ]  import nltk
     from nltk.corpus import stopwords
     from nltk.stem import WordNetLemmatizer
     nltk.download('punkt')
     nltk.download('wordnet')
     nltk.download('omw-1.4')
     nltk.download('punkt_tab')
     nltk.download('stopwords')

     lemmatizer = WordNetLemmatizer()
```

Defining the documents

```
[ ] documents = {
        "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"
    }

    custom_stopwords = {"are", "and", "not"}
```

Preprocessing the documents

```
[ ] def preprocess(text):
        tokens = nltk.word_tokenize(text.lower())
        filtered_tokens = [t for t in tokens if t.isalpha() and t not in custom_stopwords]
        lemmatized_tokens = [lemmatizer.lemmatize(t) for t in filtered_tokens]
        return lemmatized_tokens
```


Printing the preprocessed documents

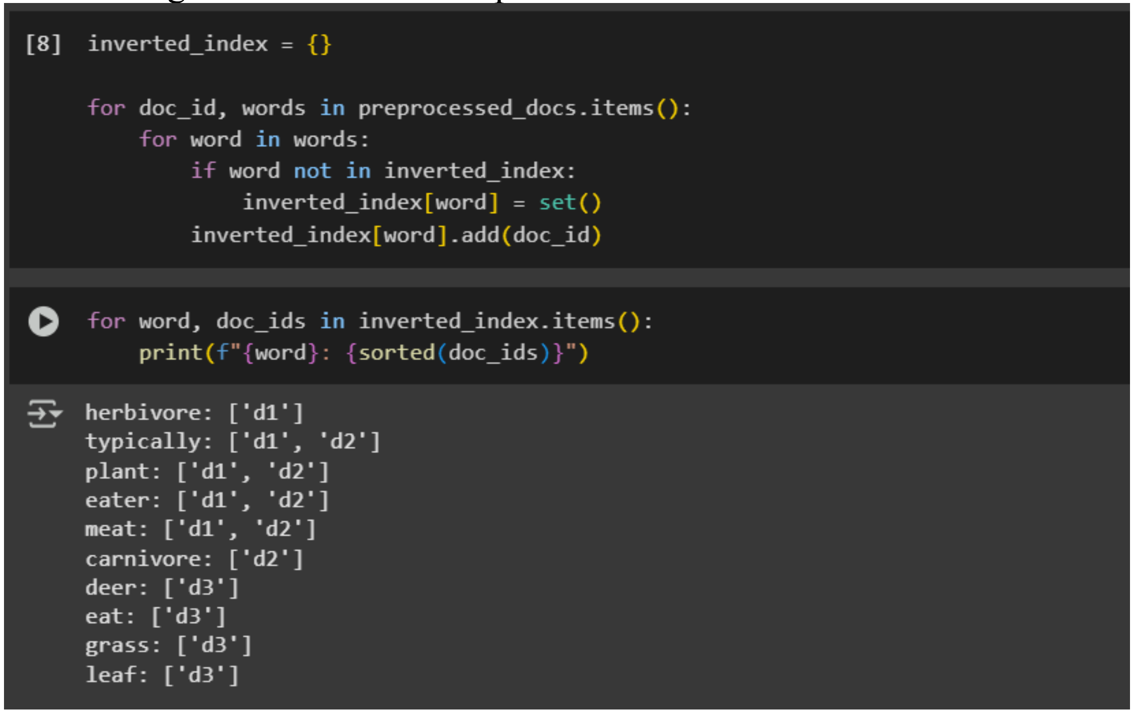
```
preprocessed_docs = {}
print("Preprocessed Documents:")
for doc_id, text in documents.items():
    preprocessed = preprocess(text)
    preprocessed_docs[doc_id] = preprocessed
    print(f"{doc_id}: {preprocessed}")
```

```
Preprocessed Documents:
d1: ['herbivore', 'typically', 'plant', 'eater', 'meat', 'eater']
d2: ['carnivore', 'typically', 'meat', 'eater', 'plant', 'eater']
d3: ['deer', 'eat', 'grass', 'leaf']
```

Output –

d1: ['herbivore', 'typically', 'plant', 'eater', 'meat', 'eater']
d2: ['carnivore', 'typically', 'meat', 'eater', 'plant', 'eater']
d3: ['deer', 'eat', 'grass', 'leaf']

For creating an Inverted index representation –



```
[8] inverted_index = {}

    for doc_id, words in preprocessed_docs.items():
        for word in words:
            if word not in inverted_index:
                inverted_index[word] = set()
            inverted_index[word].add(doc_id)
```

```
for word, doc_ids in inverted_index.items():
    print(f"{word}: {sorted(doc_ids)}")
```

```
herbivore: ['d1']
typically: ['d1', 'd2']
plant: ['d1', 'd2']
eater: ['d1', 'd2']
meat: ['d1', 'd2']
carnivore: ['d2']
deer: ['d3']
eat: ['d3']
grass: ['d3']
leaf: ['d3']
```

The inverted indexes and term frequencies of each of the tokens can be represented in a table as follows –

| Word | 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] |

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

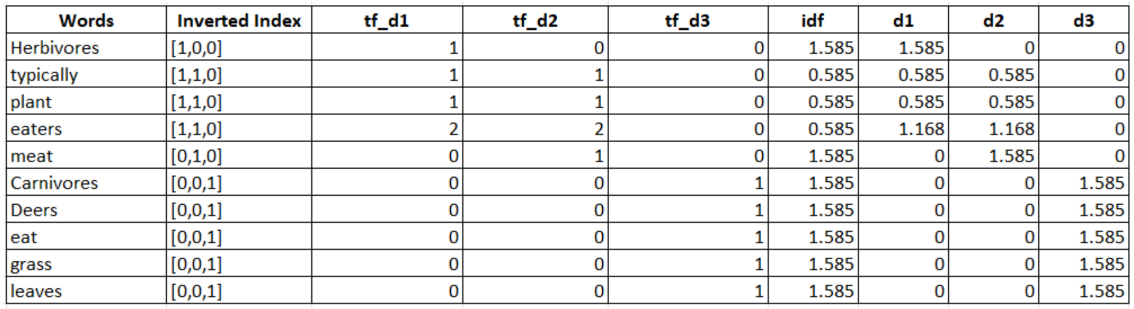
Term Frequency (TF) is determined by counting the occurrences of a word (type) within a document.
Inverse document frequency (IDF) is calculated like IDF = $log_2(N/n)$;
Where
N = total number of documents
n = the number of documents containing a given word (type)

TF-IDF helps us represent documents as vectors in a word-space, highlighting the importance of specific terms within each document.
It is calculated by using the following formula –

$$\mathrm{d}_i = \sum_{j=1}^{k} \mathrm{tf}_{d_i}(j) \cdot \mathrm{idf}(j) \cdot \mathrm{word}(j)$$

Here, di represents ith document, where j iterates over different types, and k represents the total number of types

**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 | 1.168 | 1.168 | 0 |
| meat | [0,1,0] | 0 | 1 | 0 | 1.585 | 0 | 1.585 | 0 |
| Carnivores | [0,0,1] | 0 | 0 | 1 | 1.585 | 0 | 0 | 1.585 |
| 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 |

Finally, TF-IDF matrix for each document –

d1 = [1.585, 0.585, 0.585, 1.17, 0.585, 0, 0, 0, 0, 0]

d2 = [0, 0.585, 0.585, 1.17, 0.585, 1.585, 0, 0, 0, 0]

d3 = [0, 0, 0, 0, 0, 0, 1.585, 1.585, 1.585, 1.585]

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

When using TF-IDF (Term Frequency–Inverse Document Frequency) for document retrieval, documents are selected based on the presence of non-zero TF-IDF scores for the query terms (indicating relevance)

- A query for **“plant”** will retrieve **document d1** and **document d2**, as both contain this term.
- Similarly, a query for **“eaters”** will also return **document d1** and **document d2**.

Hence, a combined query like **“plant eaters”** will return the **union** of the documents retrieved by each individual term. In this case, both **d1** and **d2** will be retrieved as they contain at least one of the query terms.

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

**Cosine Similarity calculations:**
The TF-IDF matrix for the query “plant eaters” can be represented as :

Q = [0, 0, 0.585, 0.585, 0, 0, 0, 0, 0, 0]

The cosine similarity is calculated as:

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

**Magnitude of the document and query can be calculated as follows –**

Document d1:
$\|d_1\| = \sqrt{(1.585 + 0.585 + 0.585 + 1.170 + 0.585)}$
$\|d_1\| = \sqrt{4.51} \approx$ **2.2154**
Document d2:
$\|d_2\| = \sqrt{(0.585 + 0.585 + 1.170 + 0.585 + 1.585)}$
$\|d_2\| = \sqrt{4.51} \approx$ **2.2154**
Document d3:
$\|d_3\| = \sqrt{(1.585 + 1.585 + 1.585 + 1.585 + 1.585)}$
$\|d_3\| = \sqrt{7.925} \approx$ **3.5442**
Query Q:
$\|Q\| = \sqrt{(0.585 + 0.585)}$
$\|Q\| = \sqrt{1.17} \approx$ **0.8273**

**Dot Product Between Documents and Query**
**$Q \cdot d_1$** = 0.585 + 0.585 = **1.17**
**$Q \cdot d_2$** = 0.585 + 0.585 = **1.17**
**$Q \cdot d_3$** = **0**

Cosine Similarity = $(Q \cdot d) / (\|Q\| \times \|d\|)$

- **Score of document d1 with Query Q:**
  $(1.17) / (0.8273 \times 2.2154) \approx$ **0.3734**
- **Score of document d2 with Query Q:**
  $(1.17) / (0.8273 \times 2.2154) \approx$ **0.3734**
- **Score of document d3 with Query Q:**
  $(0) / (0.8273 \times 3.5442) =$ **0**

**Ranking documents:**
The higher the cosine similarity score, the more relevant the document is

to the given query.
In this case, Documents **d1** and **d2** have the same cosine similarity score, indicating equal relevance to the query.
Document d3, on the other hand, has the lowest similarity score, suggesting minimal or no relevance.
Based on the cosine similarity values, the documents can be ranked as follows:
**Rank 1: d1, d2**
**Rank 2: d3**

**Is the ordering desirable? If no, why not?:**
No, the current ranking is not desirable due to several limitations that affect its effectiveness and accuracy:

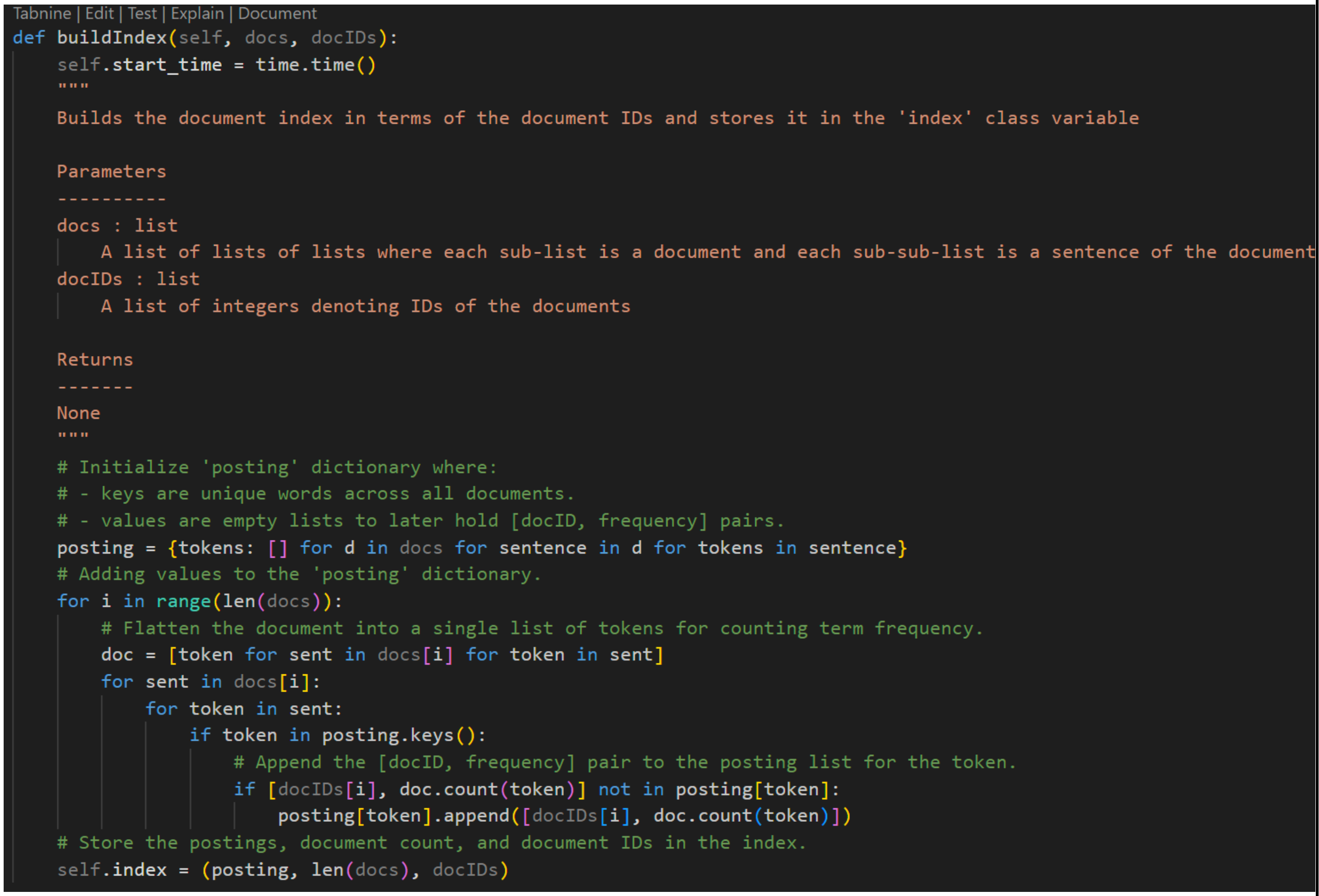
- When multiple documents receive identical cosine similarity scores, it creates ambiguity in determining which one should be prioritized. This lack of distinction reduces the precision of document ranking.
- The system heavily depends on the presence of exact query terms within documents. As a result, relevant documents that use synonymous or related terms are excluded. For example, Document d3 talks about "deers," "grass," and "leaves" — all relevant to "plant eaters" — yet it is overlooked because the exact terms are missing.
- Documents may be marked as relevant simply because they contain the query terms, regardless of their actual meaning. For instance, Document d2 discusses "carnivores," which is unrelated to the idea of "plant eaters," but is still ranked highly due to word overlap.
- The retrieval system is based solely on lexical (word-level) similarity, ignoring deeper semantic relationships. It fails to understand the meaning behind the words, which is crucial for accurate information retrieval.
- The system does not account for variations in language, such as synonyms or paraphrased expressions. A query using the phrase "vegetarian animals" would not retrieve documents about "plant eaters," even though they are conceptually identical.

## [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.

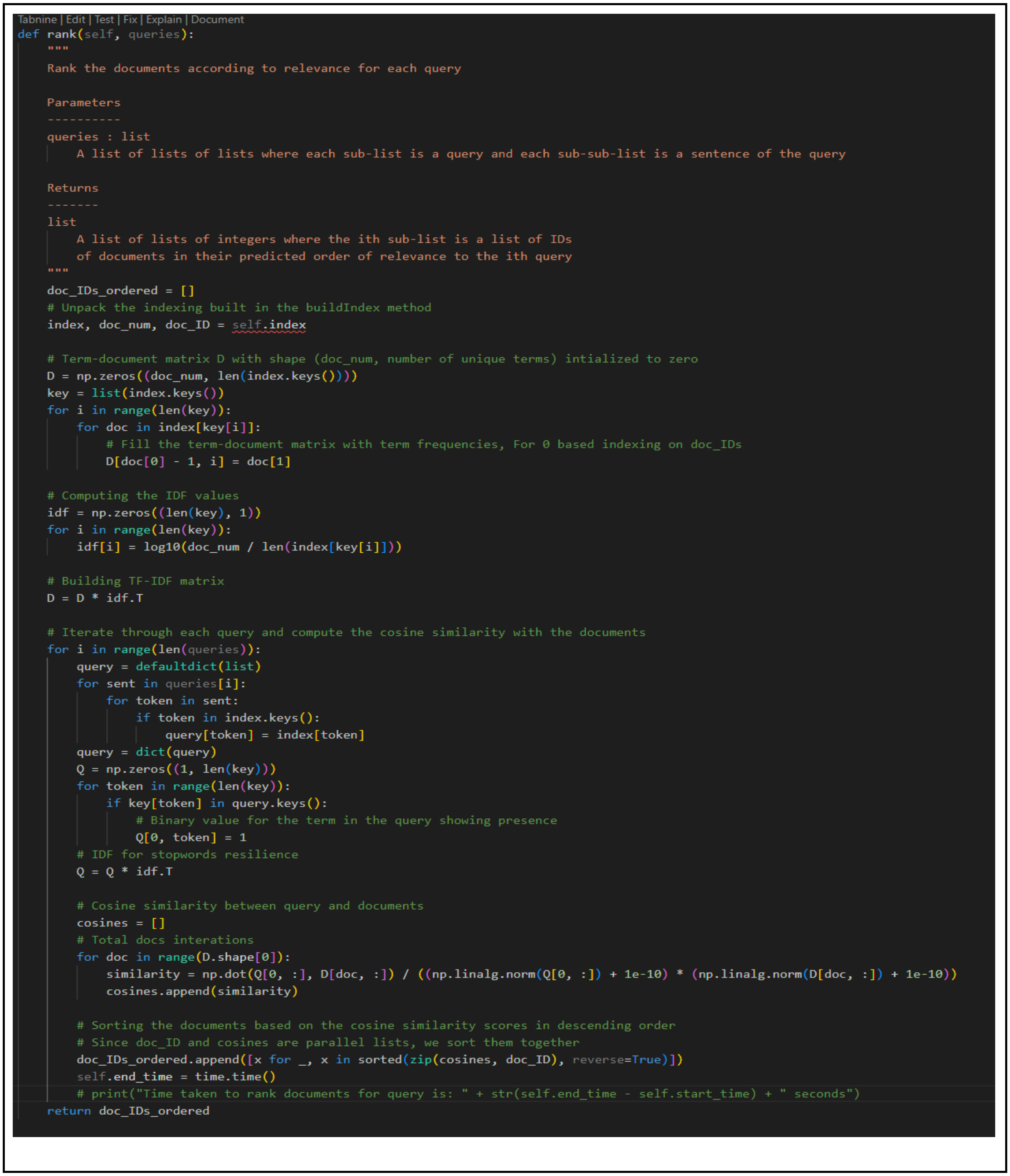
The retrieval component of the IR system takes a corpus of N documents and a set of queries, represents both documents and queries in a shared TF–IDF vector space, then ranks documents by cosine similarity to each query.
The code is implemented in a Python file named **"informationRetrieval.py".**

**Class 1: buildIndex**



```python
def buildIndex(self, docs, docIDs):
    self.start_time = time.time()
    """
    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
    """
    # Initialize 'posting' dictionary where:
    # - keys are unique words across all documents.
    # - values are empty lists to later hold [docID, frequency] pairs.
    posting = {tokens: [] for d in docs for sentence in d for tokens in sentence}
    # Adding values to the 'posting' dictionary.
    for i in range(len(docs)):
        # Flatten the document into a single list of tokens for counting term frequency.
        doc = [token for sent in docs[i] for token in sent]
        for sent in docs[i]:
            for token in sent:
                if token in posting.keys():
                    # Append the [docID, frequency] pair to the posting list for the token.
                    if [docIDs[i], doc.count(token)] not in posting[token]:
                        posting[token].append([docIDs[i], doc.count(token)])
    # Store the postings, document count, and document IDs in the index.
    self.index = (posting, len(docs), docIDs)
```

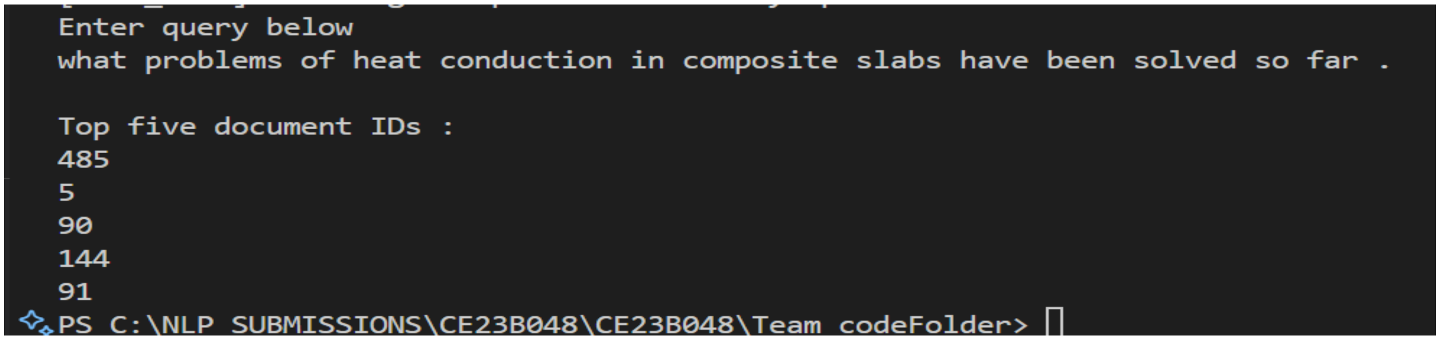
**Class 2: rank**



Tabnine | Edit | Test | Fix | Explain | Document

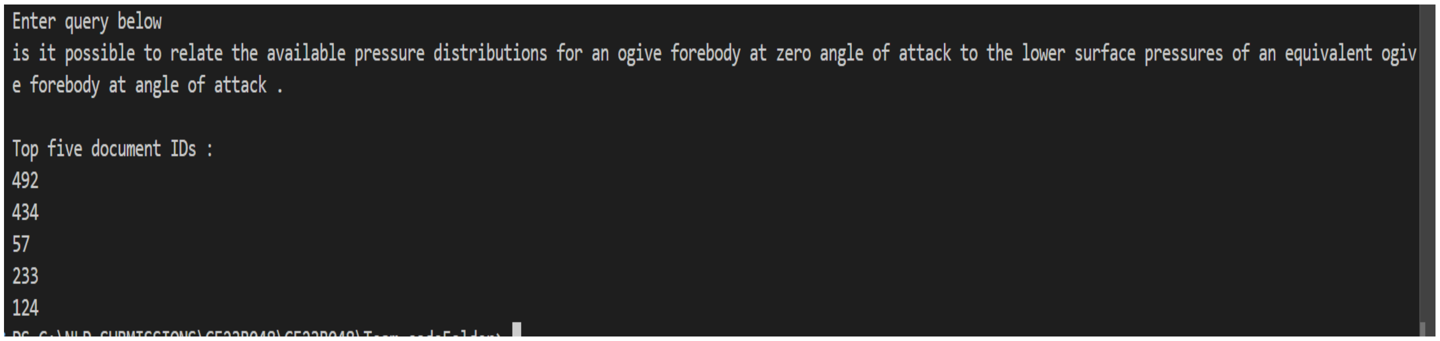
```python
def rank(self, queries):
    """
    Rank the documents according to relevance for each query

    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 = []
    # Unpack the indexing built in the buildIndex method
    index, doc_num, doc_ID = self.index

    # Term-document matrix D with shape (doc_num, number of unique terms) intialized to zero
    D = np.zeros((doc_num, len(index.keys())))
    key = list(index.keys())
    for i in range(len(key)):
        for doc in index[key[i]]:
            # Fill the term-document matrix with term frequencies, For 0 based indexing on doc_IDs
            D[doc[0] - 1, i] = doc[1]

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

    # Building TF-IDF matrix
    D = D * idf.T

    # Iterate through each query and compute the cosine similarity with the documents
    for i in range(len(queries)):
        query = defaultdict(list)
        for sent in queries[i]:
            for token in sent:
                if token in index.keys():
                    query[token] = index[token]
        query = dict(query)
        Q = np.zeros((1, len(key)))
        for token in range(len(key)):
            if key[token] in query.keys():
                # Binary value for the term in the query showing presence
                Q[0, token] = 1
        # IDF for stopwords resilience
        Q = Q * idf.T

        # Cosine similarity between query and documents
        cosines = []
        # Total docs interations
        for doc in range(D.shape[0]):
            similarity = np.dot(Q[0, :], D[doc, :]) / ((np.linalg.norm(Q[0, :]) + 1e-10) * (np.linalg.norm(D[doc, :]) + 1e-10))
            cosines.append(similarity)

        # Sorting the documents based on the cosine similarity scores in descending order
        # Since doc_ID and cosines are parallel lists, we sort them together
        doc_IDs_ordered.append([x for _, x in sorted(zip(cosines, doc_ID), reverse=True)])
        self.end_time = time.time()
        # print("Time taken to rank documents for query is: " + str(self.end_time - self.start_time) + " seconds")
    return doc_IDs_ordered
```

**Example 1:**



```
Enter query below
what problems of heat conduction in composite slabs have been solved so far .

Top five document IDs :
485
5
90
144
91
PS C:\NLP SUBMISSIONS\CE23B048\CE23B048\Team codeFolder>
```

**Example 2:**

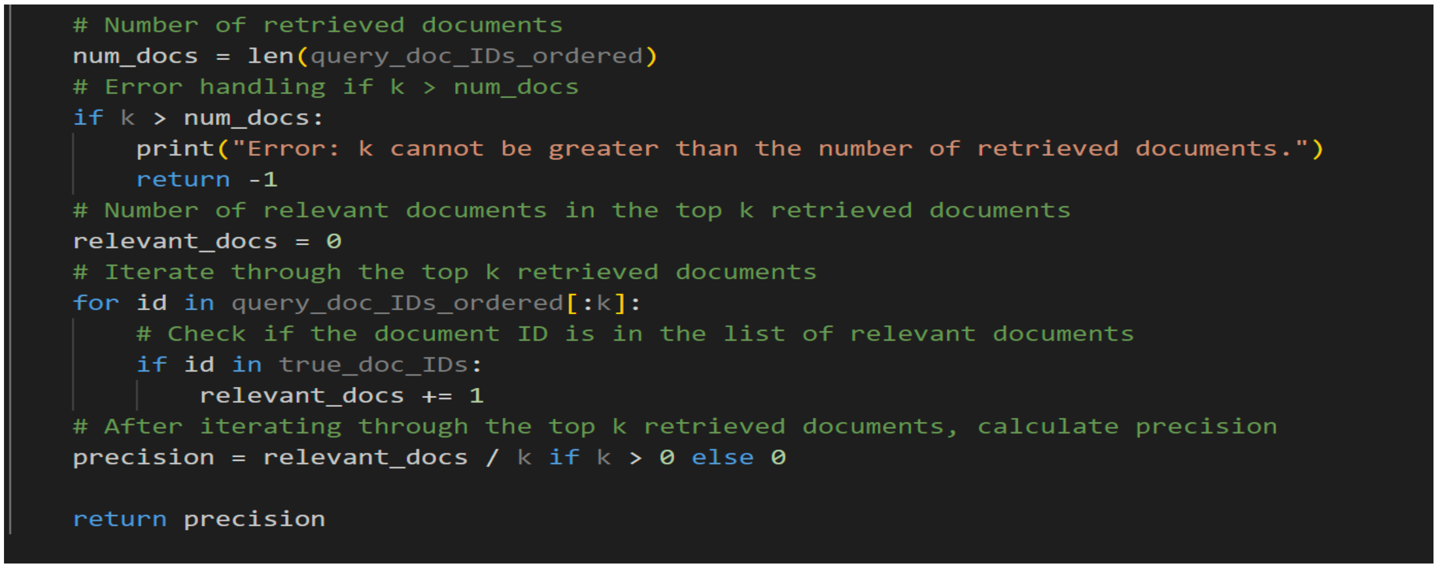


```
Enter query below
is it possible to relate the available pressure distributions for an ogive forebody at zero angle of attack to the lower surface pressures of an equivalent ogiv
e forebody at angle of attack .

Top five document IDs :
492
434
57
233
124
```

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

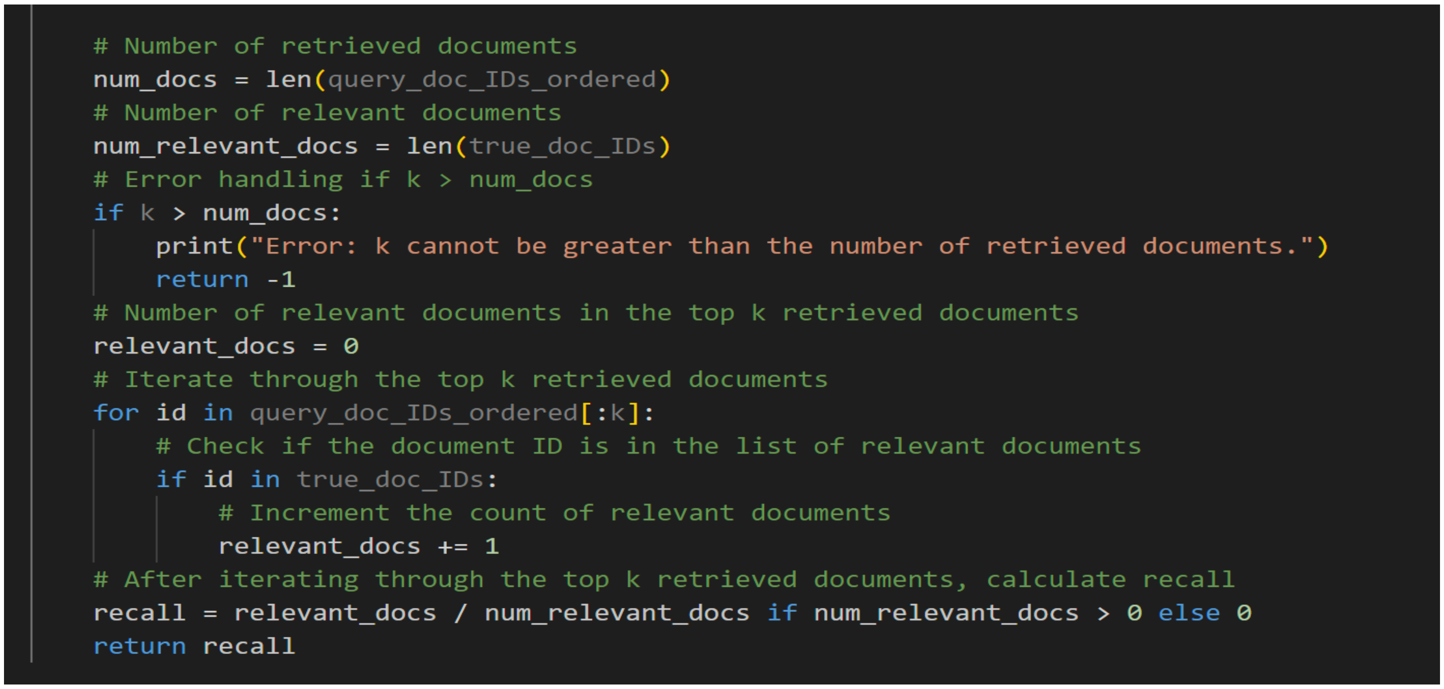
1. Implement the following evaluation measures in the template provided (i). Precision@k, (ii). Recall@k, (iii). $F_{0.5}$ score@k, (iv). AP@k, and (v) nDCG@k.

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



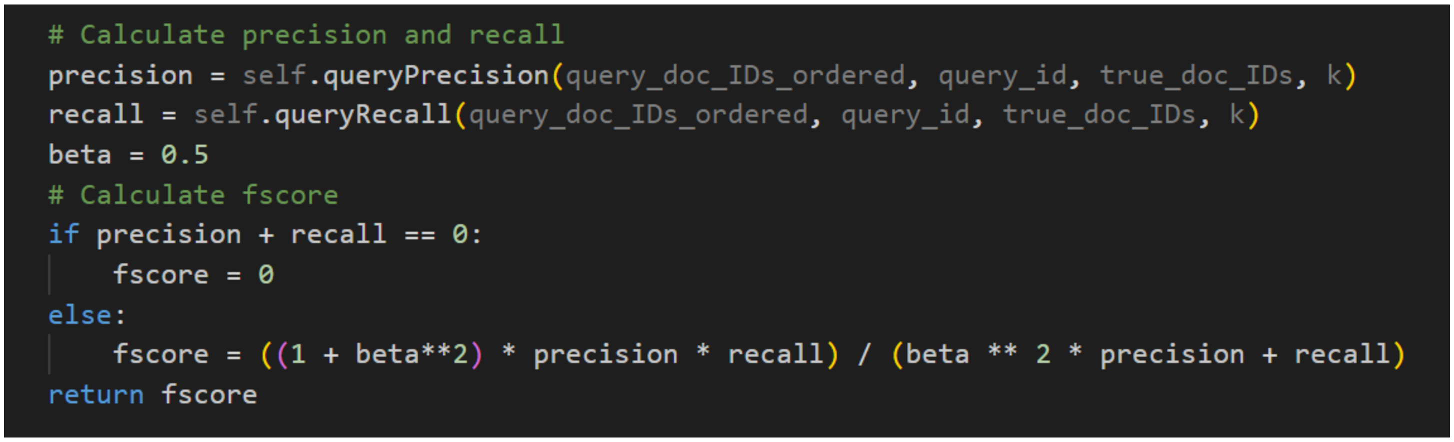
```python
# Number of retrieved documents
num_docs = len(query_doc_IDs_ordered)
# Error handling if k > num_docs
if k > num_docs:
    print("Error: k cannot be greater than the number of retrieved documents.")
    return -1
# Number of relevant documents in the top k retrieved documents
relevant_docs = 0
# Iterate through the top k retrieved documents
for id in query_doc_IDs_ordered[:k]:
    # Check if the document ID is in the list of relevant documents
    if id in true_doc_IDs:
        relevant_docs += 1
# After iterating through the top k retrieved documents, calculate precision
precision = relevant_docs / k if k > 0 else 0

return precision
```

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



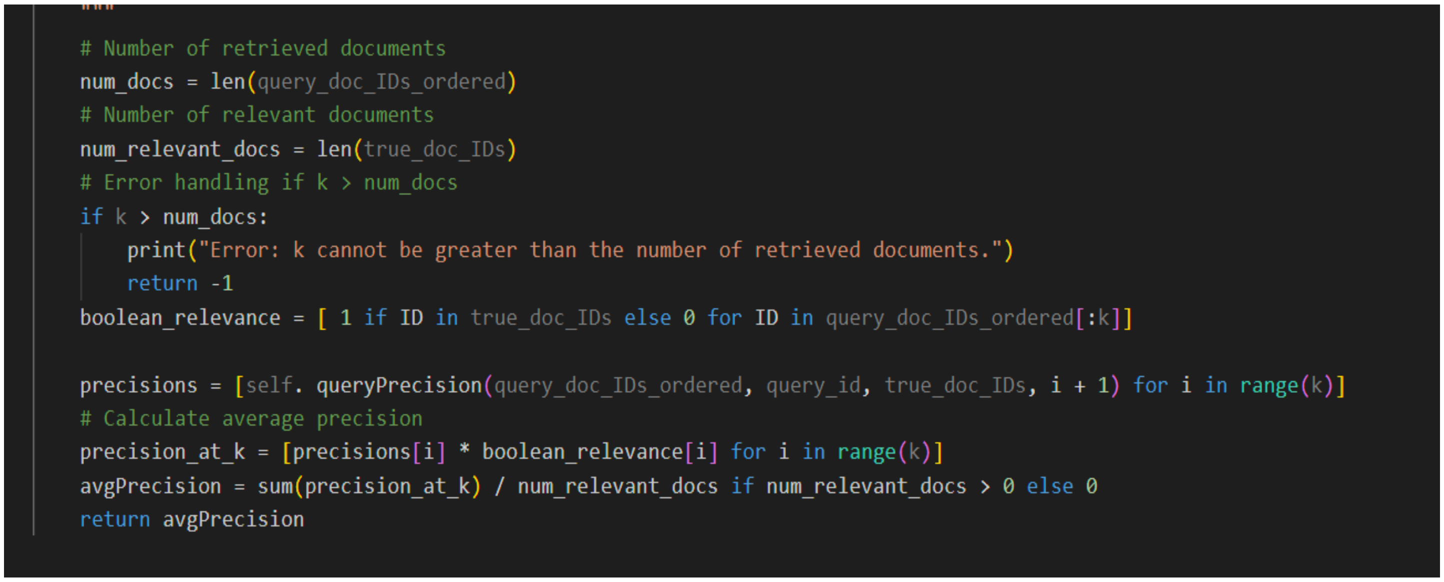
```python
# Number of retrieved documents
num_docs = len(query_doc_IDs_ordered)
# Number of relevant documents
num_relevant_docs = len(true_doc_IDs)
# Error handling if k > num_docs
if k > num_docs:
    print("Error: k cannot be greater than the number of retrieved documents.")
    return -1
# Number of relevant documents in the top k retrieved documents
relevant_docs = 0
# Iterate through the top k retrieved documents
for id in query_doc_IDs_ordered[:k]:
    # Check if the document ID is in the list of relevant documents
    if id in true_doc_IDs:
        # Increment the count of relevant documents
        relevant_docs += 1
# After iterating through the top k retrieved documents, calculate recall
recall = relevant_docs / num_relevant_docs if num_relevant_docs > 0 else 0
return recall
```

**$F_{0.5}$ score@k:** Computation of f-score of the Information Retrieval System at a given value of k for a single query



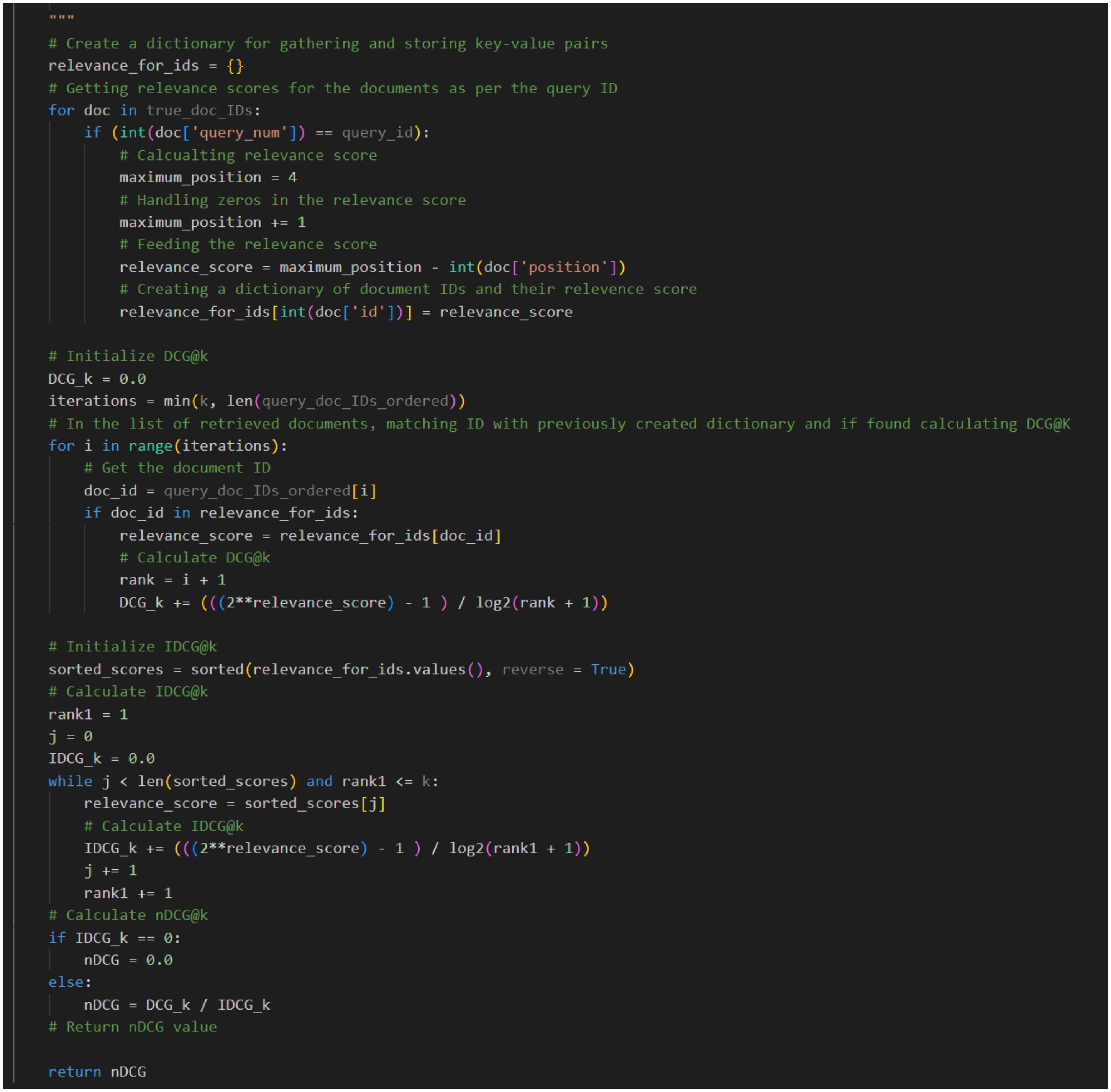
```python
# Calculate precision and recall
precision = self.queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
recall = self.queryRecall(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
beta = 0.5
# Calculate fscore
if precision + recall == 0:
    fscore = 0
else:
    fscore = ((1 + beta**2) * precision * recall) / (beta ** 2 * precision + recall)
return fscore
```

**AP@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)



```python
    """
    # Number of retrieved documents
    num_docs = len(query_doc_IDs_ordered)
    # Number of relevant documents
    num_relevant_docs = len(true_doc_IDs)
    # Error handling if k > num_docs
    if k > num_docs:
        print("Error: k cannot be greater than the number of retrieved documents.")
        return -1
    boolean_relevance = [ 1 if ID in true_doc_IDs else 0 for ID in query_doc_IDs_ordered[:k]]

    precisions = [self. queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, i + 1) for i in range(k)]
    # Calculate average precision
    precision_at_k = [precisions[i] * boolean_relevance[i] for i in range(k)]
    avgPrecision = sum(precision_at_k) / num_relevant_docs if num_relevant_docs > 0 else 0
    return avgPrecision
```

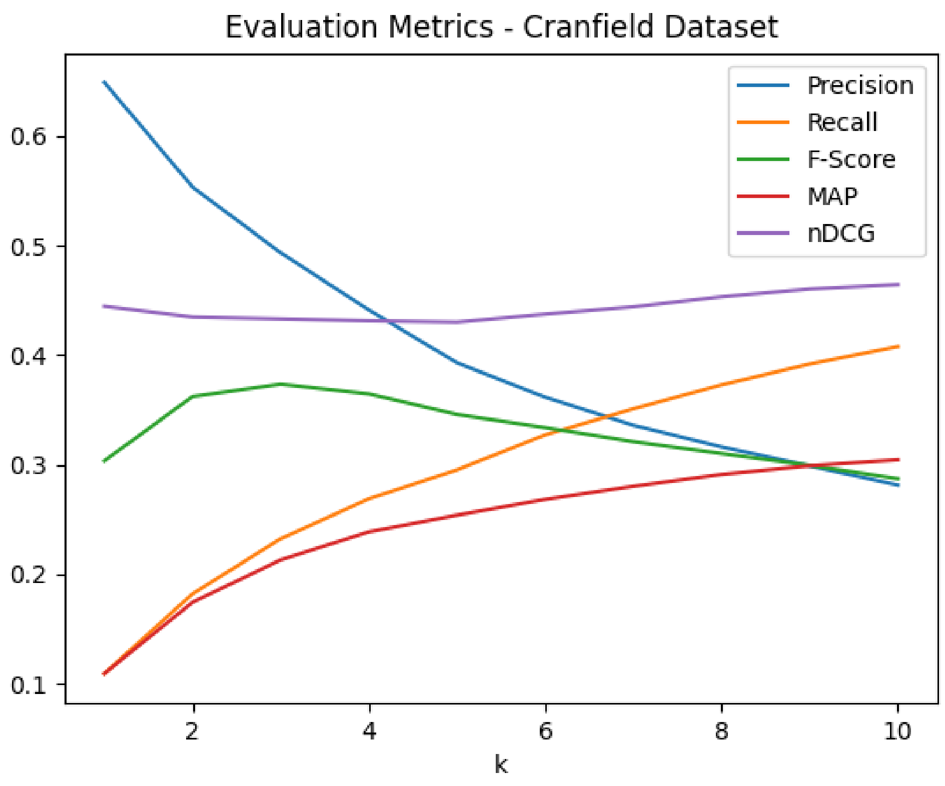
**nDCG@k:** Computation of nDCG of the Information Retrieval System at given value of k for a single query.



```
    """
    # Create a dictionary for gathering and storing key-value pairs
    relevance_for_ids = {}
    # Getting relevance scores for the documents as per the query ID
    for doc in true_doc_IDs:
        if (int(doc['query_num']) == query_id):
            # Calcualting relevance score
            maximum_position = 4
            # Handling zeros in the relevance score
            maximum_position += 1
            # Feeding the relevance score
            relevance_score = maximum_position - int(doc['position'])
            # Creating a dictionary of document IDs and their relevence score
            relevance_for_ids[int(doc['id'])] = relevance_score

    # Initialize DCG@k
    DCG_k = 0.0
    iterations = min(k, len(query_doc_IDs_ordered))
    # In the list of retrieved documents, matching ID with previously created dictionary and if found calculating DCG@K
    for i in range(iterations):
        # Get the document ID
        doc_id = query_doc_IDs_ordered[i]
        if doc_id in relevance_for_ids:
            relevance_score = relevance_for_ids[doc_id]
            # Calculate DCG@k
            rank = i + 1
            DCG_k += (((2**relevance_score) - 1 ) / log2(rank + 1))

    # Initialize IDCG@k
    sorted_scores = sorted(relevance_for_ids.values(), reverse = True)
    # Calculate IDCG@k
    rank1 = 1
    j = 0
    IDCG_k = 0.0
    while j < len(sorted_scores) and rank1 <= k:
        relevance_score = sorted_scores[j]
        # Calculate IDCG@k
        IDCG_k += (((2**relevance_score) - 1 ) / log2(rank1 + 1))
        j += 1
        rank1 += 1
    # Calculate nDCG@k
    if IDCG_k == 0:
        nDCG = 0.0
    else:
        nDCG = DCG_k / IDCG_k
    # Return nDCG value

    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:**

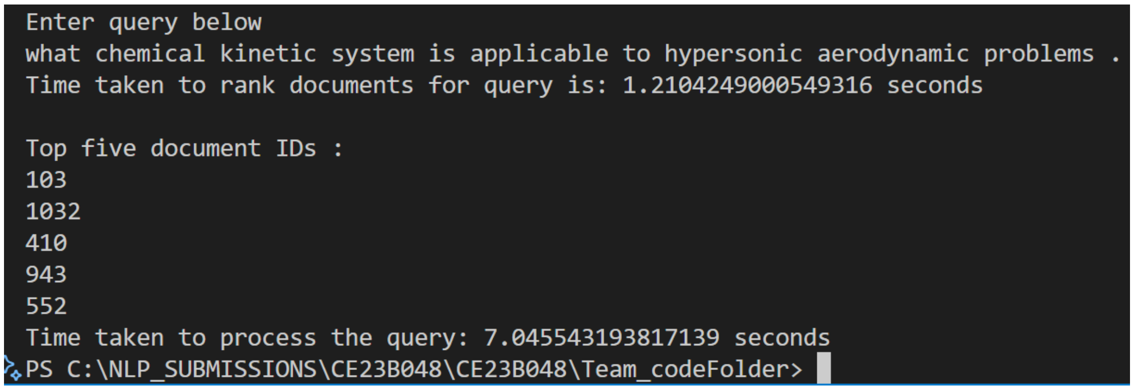


**Observation:**

**Precision@k :** As we pull in more documents (increasing k), precision usually drops. First, the top results are the most relevant, so precision is high. But when we ask for more (larger k), the system starts adding documents that might not be as useful or even completely unrelated. This dilutes the quality of the results, since the proportion of relevant documents shrinks, causing precision to fall.
**Recall@k:** When we increase k (get more documents), recall goes up or stays the same. This happens because the total number of relevant documents doesn't change, but the more documents we collect, the more chances we must find the relevant ones. So, recall never drops; it either increases or stays where it is as we get more results.

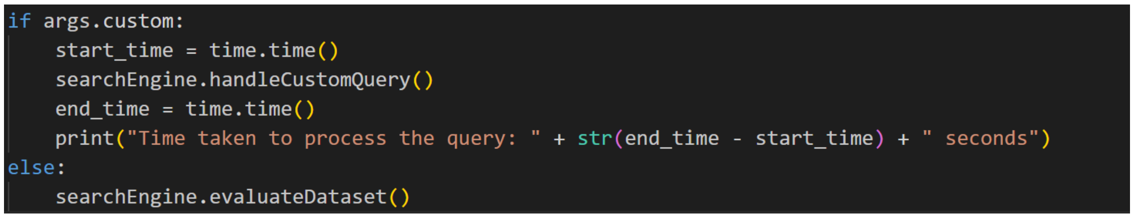
**F-score@k**: For F0.5-score, precision is weighted more heavily than recall (β = 0.5). The general shape of F0.5 will follow a similar trend to the F1-score but favor precision. Here, the precision is favored 4x times the recall. Hence, we can see both precision and F-score been nearly same at the end.
**MAP@k:** Map never decreases as we increase k; because we would be adding Precision if we find a relevant doc while increasing k. This would increase Average Precision for each query because the denominator is constant and we are adding precision at K we find relevant documents. Therefore, considering more documents generally leads to a higher average precision and thus an increase in MAP.
**nDCG@K:** Slight dip at lower k, then stabilizes around 0.44-0.46. Shows the ranking quality remains consistent, even as more documents are added.

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



```
Enter query below
what chemical kinetic system is applicable to hypersonic aerodynamic problems .
Time taken to rank documents for query is: 1.2104249000549316 seconds

Top five document IDs :
103
1032
410
943
552
Time taken to process the query: 7.045543193817139 seconds
PS C:\NLP_SUBMISSIONS\CE23B048\CE23B048\Team_codeFolder>
```

The time taken to process the query: **7.045543193817139 seconds** shows the time module wrapped around whole IR system in the following way.



```
if args.custom:
    start_time = time.time()
    searchEngine.handleCustomQuery()
    end_time = time.time()
    print("Time taken to process the query: " + str(end_time - start_time) + " seconds")
else:
    searchEngine.evaluateDataset()
```

Whereas Time taken to rank documents for query is: **1.2104249000549316 seconds** shows time module wrapped around the **InformationRetrieval** class for one custom query.

[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 and Examples from your results:

**The limitations of the Vector Space Models are -**
**1. Synonymy and Vocabulary mismatch**
A pure TF-IDF VSM only matches documents and queries on identical terms. As a result, it cannot recognize different words that express the same concept.

- **Cranfield Illustration**
    - **Query:** "heat conduction composite slabs"
    - **Document 5** discusses "transient heat **conduction** into a double-layer slab" .
    - **Document 6** examines "transient heat **flow** in a multilayer slab" .
      Although both documents address heat transfer in layered slabs, the VSM will strongly favour Document 5 which contains "conduction" and will undervalue or omit Document 6 which uses "flow" instead of "conduction", despite its equal relevance.

**2. Ignoring word proximity and phrase structure**
Because VSM treats text as an unordered "bag of words," it cannot distinguish whether query terms occur together as a coherent phrase or merely appear independently in different contexts.

- **Cranfield Illustration**
    - **Query:** "boundary layer control effect"
    - **Document 1** explicitly studies the "destalling or **boundary-layer-control effect**" .
    - Another document may mention "boundary" once and "effect" elsewhere without addressing the combined phenomenon.
      Under VSM, both documents accrue identical term match scores, yet only Document 1 actually investigates the precise effect of interest.

**3. Document length bias**
Longer documents naturally contain more term occurrences and thus tend to achieve higher TF-IDF dot products, unless additional length normalization like that of BM25 adjustments is applied.

- **Cranfield Illustration**
    - **Document 21** "on heat transfer in slip flow" is a concise, focused study .

**Document 37** "the prospects for magneto-aerodynamics" is substantially longer and merely mentions "heat," "transfer," and "flow" in passing .
In a raw VSM, Document 37 may outrank Document 21 solely by virtue of its greater term frequency, despite Document 21's greater topical relevance.

**4. Polysemy and ambiguity**
Single words with multiple senses contribute equally to similarity scores, yet only some occurrences reflect the intended query sense.

- **Cranfield Illustration**
    - The term "plate" appears in **Document 2** to describe a "flat plate in shear flow" and in **Document 29** as part of a thermal stress model .
    - **Query:** "boundary plate flow"
      VSM cannot distinguish which sense of "plate" the user intends, and may retrieve both documents with similar scores, even though only Document 2 addresses boundary layer flow.

**5. Absence of conceptual or semantic relationships**
VSM does not capture hierarchical or ontological relationships. For example, that "wing" is a special case of "aerodynamic surface" and nor does it generalize across related terms.

- **Cranfield Illustration**
    - **Query:** "aerodynamic surfaces in slipstream"
    - **Document 1** studies "the aerodynamics of a wing in a slipstream" .
      If one were to phrase the query using "lifting surface" instead of "wing," the VSM would fail entirely to retrieve Document 1, despite its conceptual match.

**6. Zero weight for rare or out-of-vocabulary terms**
Highly discriminative terms that occur very infrequently or that are omitted during indexing receive zero weight, preventing retrieval of the one document in which they matter.

- **Cranfield Illustration**
    - The term "destalling" appears uniquely in **Document 1** .
    - **Query:** "destalling boundary layer"
      If "destalling" were inadvertently excluded from the index through hysterical preprocessing, VSM would be unable to retrieve Document 1, which is its precisely relevant match.

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.