CS6370: Natural Language Processing
Project

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

Name:	Roll No.:
SANJEEV M	BE21B034
SHUBAN V	BS21B032
VISHWANATH VINOD	EE22B002
PRITHVI P RAO	EE22B024
JAANAV MATHAVAN	EE22B167

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.

Case (i):
Assuming we need to segment (in this case no individual sentences), tokenize, reduce to root form before removing the stopwords. *(IR preprocessing)*
This method involves the following steps:

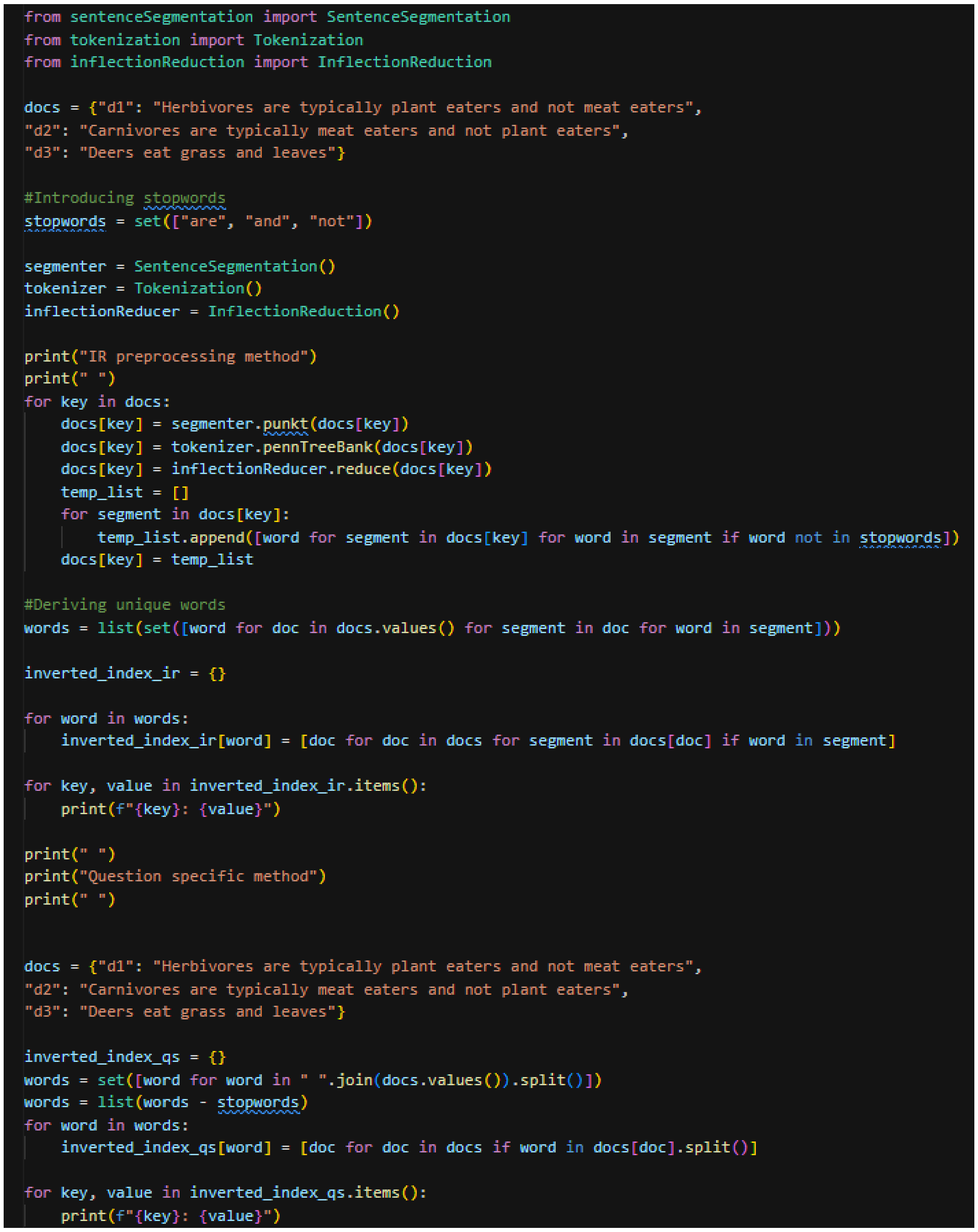
1. Sentence segmentation
2. Tokenization using Penn Treebank
3. Stemming - Reducing to root form (Inflexion)
4. Removing stopwords
5. Determining Inverted index for every unique word

```
1. deer: ['d3']
2. plant: ['d1', 'd2']
3. meat: ['d1', 'd2']
4. typic: ['d1', 'd2']
5. eater: ['d1', 'd2']
6. leav: ['d3']
7. eat: ['d3']
8. grass: ['d3']
9. carnivor: ['d2']
10.herbivor: ['d1']
```

Case (ii):
Considering individual words in the docs as one word without considering root form. Moreover there is only one sentence in each doc. *(Question - specific)*
This method involves the following steps:

1. Determining unique words in the doc
2. Removing stopwords
3. Inverted indexing of the result words

```
1. Herbivores: ['d1']
2. leaves: ['d3']
3. plant: ['d1', 'd2']
4. Carnivores: ['d2']
5. meat: ['d1', 'd2']
6. typically: ['d1', 'd2']
7. eaters: ['d1', 'd2']
8. eat: ['d3']
9. grass: ['d3']
10.Deers: ['d3']
```

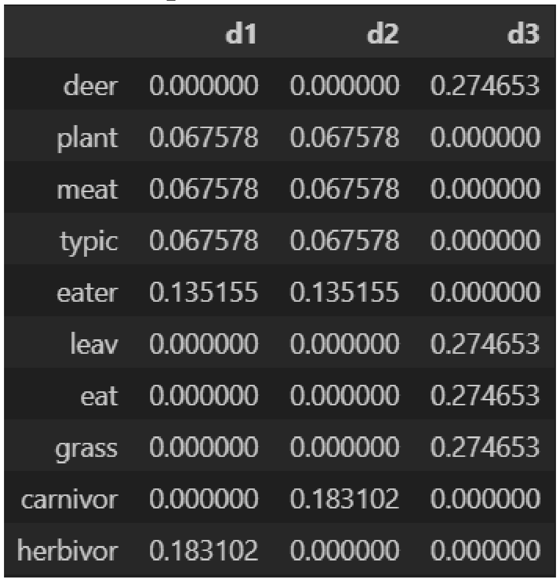


```python
from sentenceSegmentation import SentenceSegmentation
from tokenization import Tokenization
from inflectionReduction import InflectionReduction

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"}

#Introducing stopwords
stopwords = set(["are", "and", "not"])

segmenter = SentenceSegmentation()
tokenizer = Tokenization()
inflectionReducer = InflectionReduction()

print("IR preprocessing method")
print(" ")
for key in docs:
    docs[key] = segmenter.punkt(docs[key])
    docs[key] = tokenizer.pennTreeBank(docs[key])
    docs[key] = inflectionReducer.reduce(docs[key])
    temp_list = []
    for segment in docs[key]:
        temp_list.append([word for segment in docs[key] for word in segment if word not in stopwords])
    docs[key] = temp_list

#Deriving unique words
words = list(set([word for doc in docs.values() for segment in doc for word in segment]))

inverted_index_ir = {}

for word in words:
    inverted_index_ir[word] = [doc for doc in docs for segment in docs[doc] if word in segment]

for key, value in inverted_index_ir.items():
    print(f"{key}: {value}")

print(" ")
print("Question specific method")
print(" ")


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"}

inverted_index_qs = {}
words = set([word for word in " ".join(docs.values()).split()])
words = list(words - stopwords)
for word in words:
    inverted_index_qs[word] = [doc for doc in docs if word in docs[doc].split()]

for key, value in inverted_index_qs.items():
    print(f"{key}: {value}")
```

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

```
[[0.         0.         0.27465307]
 [0.06757752 0.06757752 0.        ]
 [0.06757752 0.06757752 0.        ]
 [0.06757752 0.06757752 0.        ]
 [0.13515504 0.13515504 0.        ]
 [0.         0.         0.27465307]
 [0.         0.         0.27465307]
 [0.         0.         0.27465307]
 [0.         0.18310205 0.        ]
 [0.18310205 0.         0.        ]]
```

This is the tf-idf term document matrix with rows corresponding to the terms and columns to documents.
Below is a pandas DataFrame with rows & column names.



	d1	d2	d3
deer	0.000000	0.000000	0.274653
plant	0.067578	0.067578	0.000000
meat	0.067578	0.067578	0.000000
typic	0.067578	0.067578	0.000000
eater	0.135155	0.135155	0.000000
leav	0.000000	0.000000	0.274653
eat	0.000000	0.000000	0.274653
grass	0.000000	0.000000	0.274653
carnivor	0.000000	0.183102	0.000000
herbivor	0.183102	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 for the query “plant eaters”, once the query is tokenized and stemmed to its root form, we will have plant and eater as the terms of the query. So based on inverted indexing, ‘plant’ will be mapped to docs = {‘d1’, ‘d2’} and ‘eater’ will be mapped to docs = {‘d1’, ‘d2’}. Based on the inverted index constructed, docs {‘d1’, ‘d2’} are retrieved as we will need to take the intersection and we will end up with this.

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

Cosine Similarity calculations:

Cosine similarities : `[0.56015692 0.56015692 0.        ]`

Ranking documents:

Rankings:

```
   1. Document d2: 0.560156917515788
   2. Document d1: 0.560156917515788
   3. Document d3: 0.0
```

Is the ordering desirable? If no, why not?:

The ordering of the results is not desirable. The stopword removal process includes words like "not", which can drastically alter the meaning of a sentence. For instance, documents "d1" and "d2" are complete opposites in meaning, but even if "not" were not removed, we would likely still get the same similarity scores for “d1” and “d2”.

```
   1. Document d2: 0.5415929869271332
   2. Document d1: 0.5415929869271332
   3. Document d3: 0.0
```

When it comes to the query *"plant eaters"*, we would expect the output to favor "d1". However, since TF-IDF considers only the frequency and presence of terms in the document without taking context, word order, or co-occurrence into account—it assigns equal similarity scores to semantically opposite sentences. This results in the incorrect or non-desirable rankings in the output.

[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.

Updated the template with the code for the IR component. We have the following methods:

   1. **buildIndex**: we need to build the inverted index with each unique

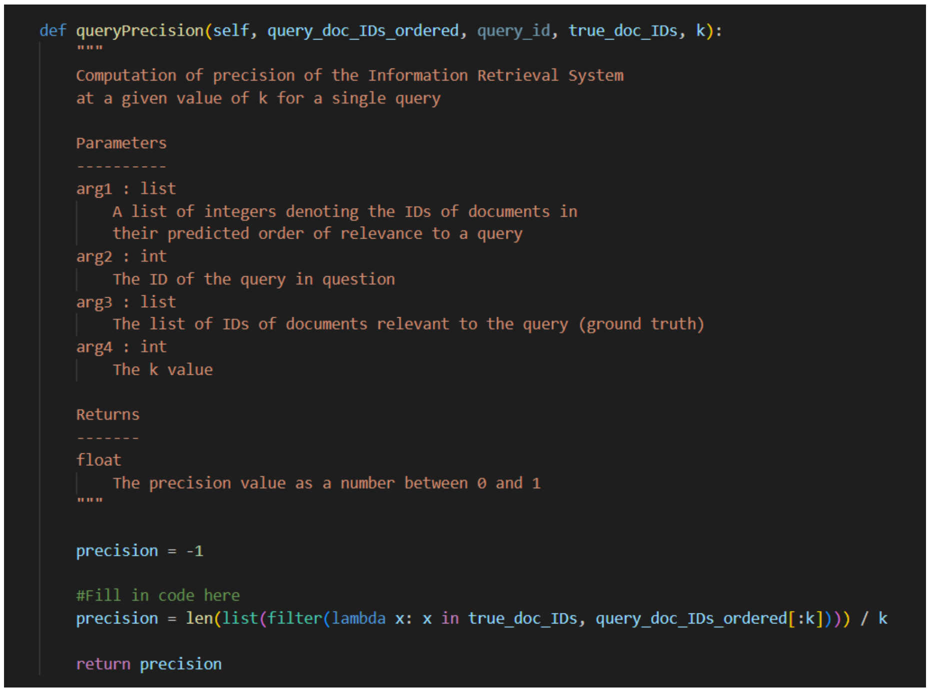
term mapped to the corresponding documents in a dictionary.

2. **compute_tf_idf_matrix** : This method takes in the documents and docIDs and it computes the tf_idf matrix for the documents and updates the idf component for each word. (This will throw error if the query contains words not present in the cranfield corpus).
3. **rank** : Gets the tf-idf matrix of the documents and uses the idf component of each word to compute the query vector. From the stems (root words) of the query, we determine the documents to refer to using the inverted index in order to reduce complexity. This will not be able to capture synonyms or contextually same content. It requires same words to be present in the document as in the query. Then using the similarity between the reduced documents and the query vector we rank the documents for retrieval.

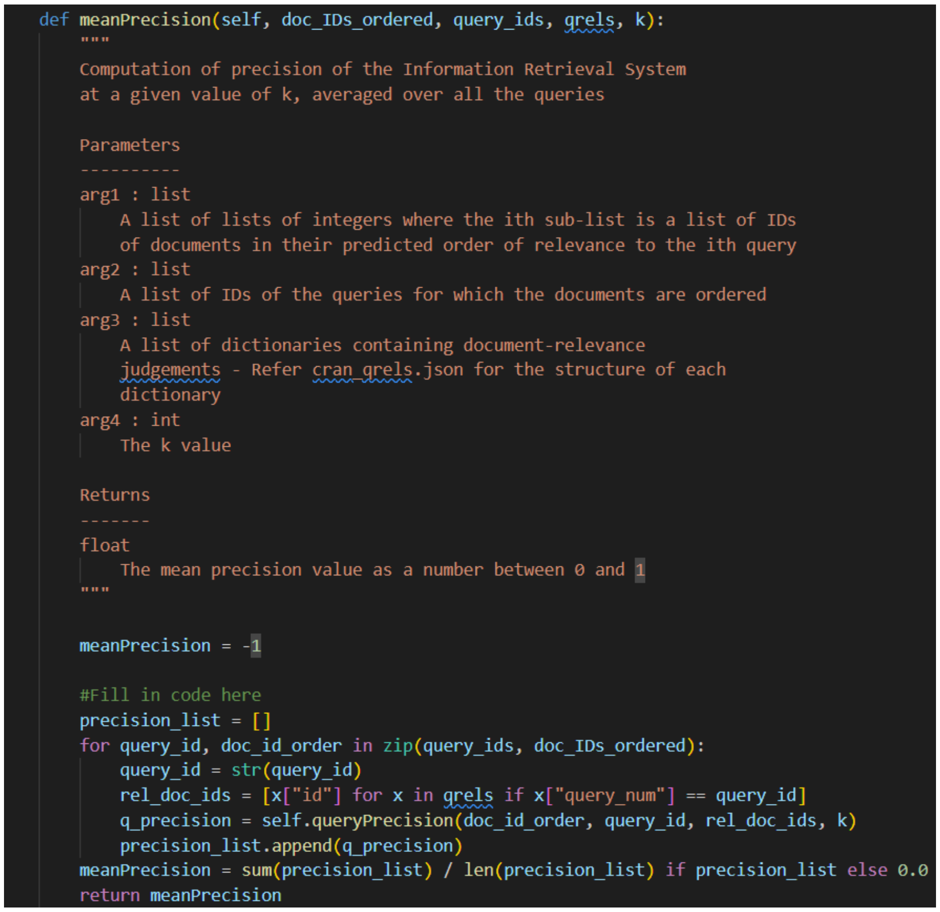
[Warm up] Part 3: Evaluating your IR system [Implementation]

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:

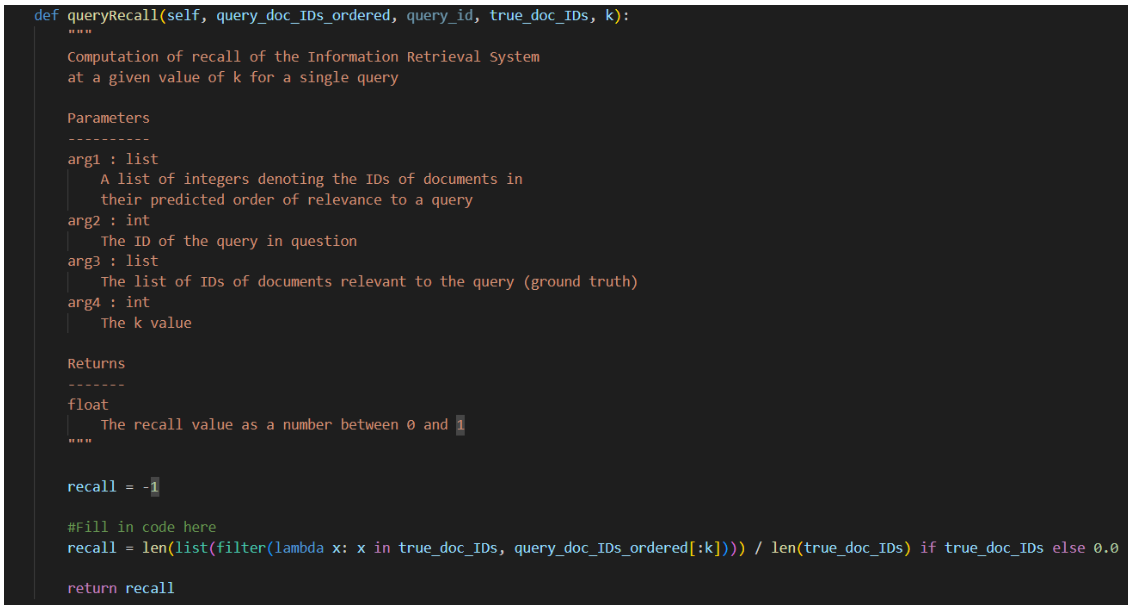


```python
def queryPrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    """
    Computation of precision of the Information Retrieval System
    at a given value of k for a single query

    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
    precision = len(list(filter(lambda x: x in true_doc_IDs, query_doc_IDs_ordered[:k]))) / k

    return precision
```

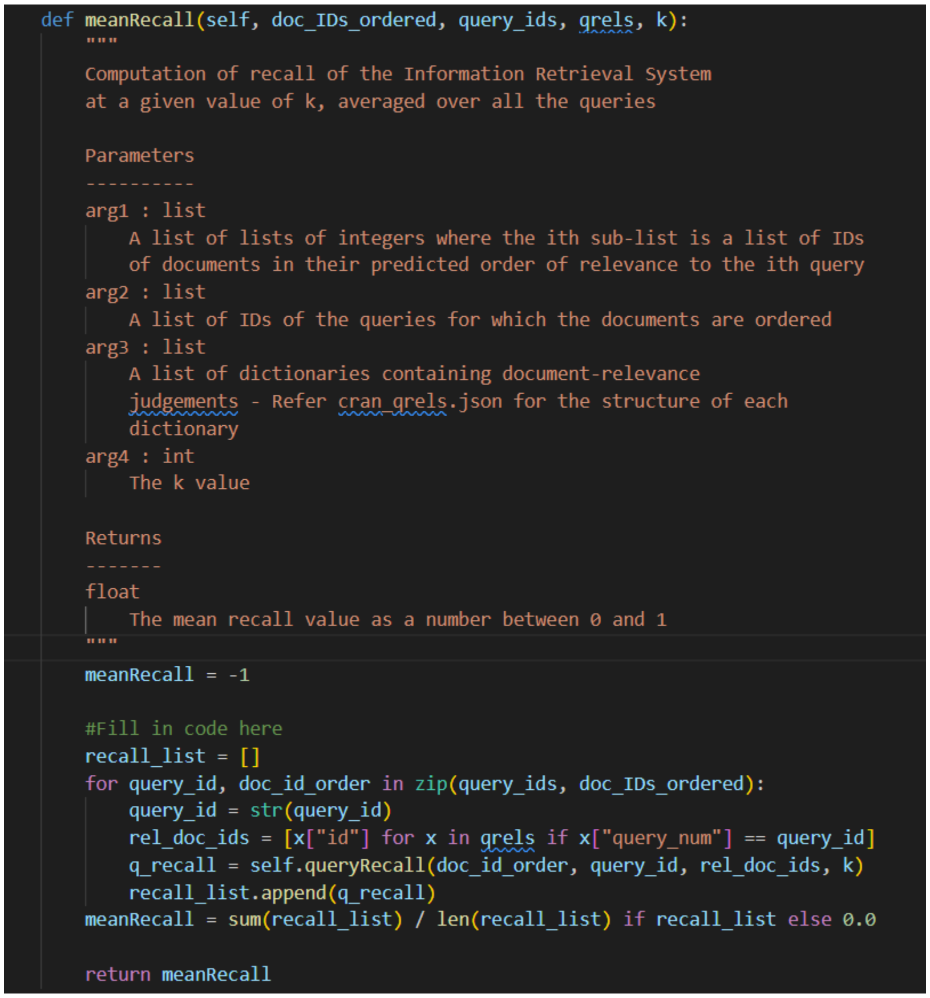


```python
def meanPrecision(self, doc_IDs_ordered, query_ids, qrels, k):
    """
    Computation of precision of the Information Retrieval System
    at a given value of k, averaged over all the queries

    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
    precision_list = []
    for query_id, doc_id_order in zip(query_ids, doc_IDs_ordered):
        query_id = str(query_id)
        rel_doc_ids = [x["id"] for x in qrels if x["query_num"] == query_id]
        q_precision = self.queryPrecision(doc_id_order, query_id, rel_doc_ids, k)
        precision_list.append(q_precision)
    meanPrecision = sum(precision_list) / len(precision_list) if precision_list else 0.0
    return meanPrecision
```

Recall@k:

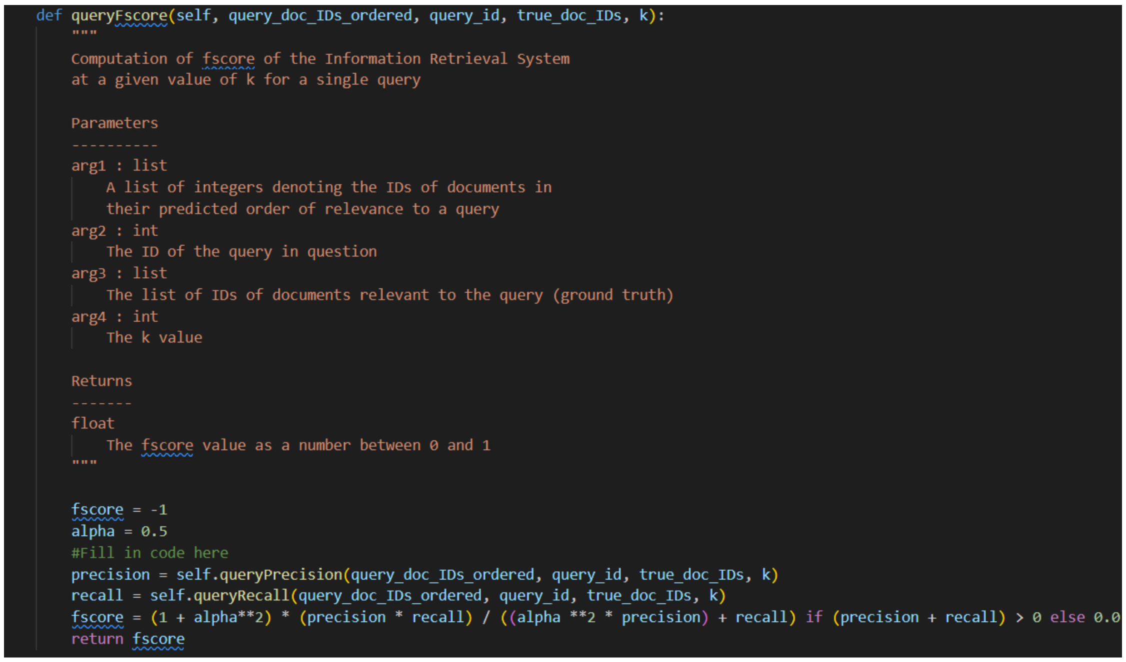


```python
    def queryRecall(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        """
        Computation of recall of the Information Retrieval System
        at a given value of k for a single query

        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
        recall = len(list(filter(lambda x: x in true_doc_IDs, query_doc_IDs_ordered[:k]))) / len(true_doc_IDs) if true_doc_IDs else 0.0

        return recall
```

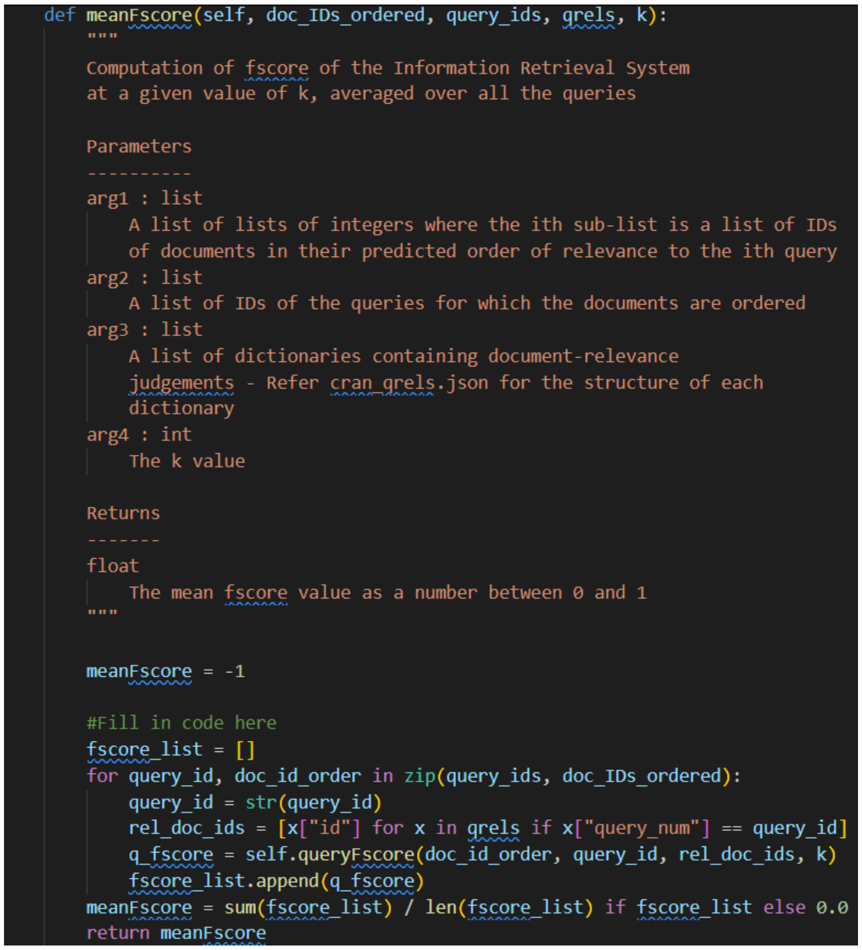


```python
    def meanRecall(self, doc_IDs_ordered, query_ids, qrels, k):
        """
        Computation of recall of the Information Retrieval System
        at a given value of k, averaged over all the queries

        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
        recall_list = []
        for query_id, doc_id_order in zip(query_ids, doc_IDs_ordered):
            query_id = str(query_id)
            rel_doc_ids = [x["id"] for x in qrels if x["query_num"] == query_id]
            q_recall = self.queryRecall(doc_id_order, query_id, rel_doc_ids, k)
            recall_list.append(q_recall)
        meanRecall = sum(recall_list) / len(recall_list) if recall_list else 0.0

        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
        alpha = 0.5
        #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)
        fscore = (1 + alpha**2) * (precision * recall) / ((alpha **2 * precision) + recall) if (precision + recall) > 0 else 0.0
        return fscore
```

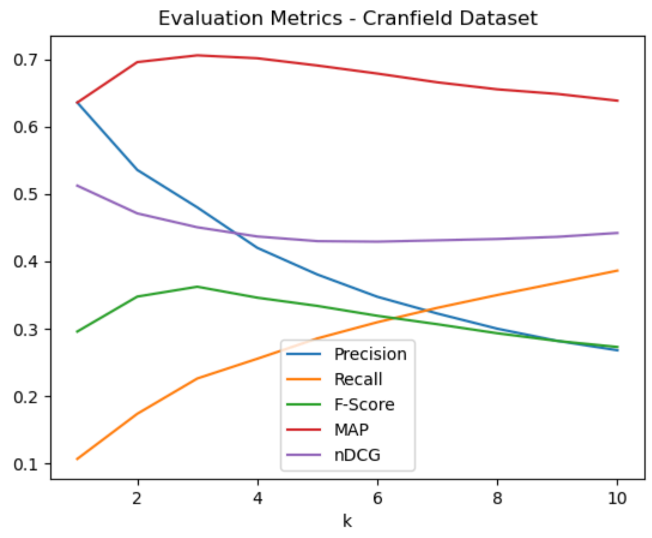


```python
    def meanFscore(self, doc_IDs_ordered, query_ids, qrels, k):
        """
        Computation of fscore of the Information Retrieval System
        at a given value of k, averaged over all the queries

        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
        fscore_list = []
        for query_id, doc_id_order in zip(query_ids, doc_IDs_ordered):
            query_id = str(query_id)
            rel_doc_ids = [x["id"] for x in qrels if x["query_num"] == query_id]
            q_fscore = self.queryFscore(doc_id_order, query_id, rel_doc_ids, k)
            fscore_list.append(q_fscore)
        meanFscore = sum(fscore_list) / len(fscore_list) if fscore_list else 0.0
        return meanFscore
```

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 rank increases
2. Recall increases as rank increases.
3. $F_{0.5}$-score increases initially but later on decreases but passes close to the point of intersection of precision and recall curves
4. MAP remains at the top as it gives importance to the first document & demands it to be relevant. As our Vector space model predicts the first document in most cases correctly, we observe that the MAP remains higher. There is an increase for lower values of k but later there is a slight dip.
5. nDCG as mostly in plateau phase, It dips slightly initially and remains almost constant with a very negligible rise towards higher ranks.

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

Cranfield Query Dataset Time taken
- Custom code built from scratch: 388.384 seconds (388s - 6 min 28 sec)
- tf-idf vectorizer from sklearn feature extraction module : 7.76 sec

Custom Query Time taken
- Custom code built from scratch: 388.384 seconds (388s - 6 min 28 sec)
- tf-idf vectorizer from sklearn feature extraction module : 4.90 sec

[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:

- Computationally expensive to reevaluate the vectors if new term is added in the doc. (E.g: It took too much time when we wrote in scratch and above answers have evidence)
- Order of the words is ignored. (When we gave 2 custom queries we got the same 5 documents)
- Large documents – difficult similarity. (Overcome by BM25 - accuracy improved after its usage)
- False Negatives: due to synonymity (diff words, same meaning). (We tried using Query expansion - Wordnet synsets)
- False positives: due to wrong words, suffix & prefix removal mistakes. (We tried to use LSA, ESA (trained on aerospace engineering concepts to overcome the false positives)
- Semantic content ⇒ not inclusive (We used Dense embeddings to check if we can overcome this problem)

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.