CS6370: Natural Language Processing
Project

Release Date: 21st March 2024 Deadline: 15th May

Name:	Roll No.:
Sheth Jenil Samir	BE20B031
Meghana Banoth	CS21B014
Himakar Sai Dasari	CS21B022

General Instructions:

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

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

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

Consider the following three documents:

d_1: Herbivores are typically plant eaters and not meat eaters

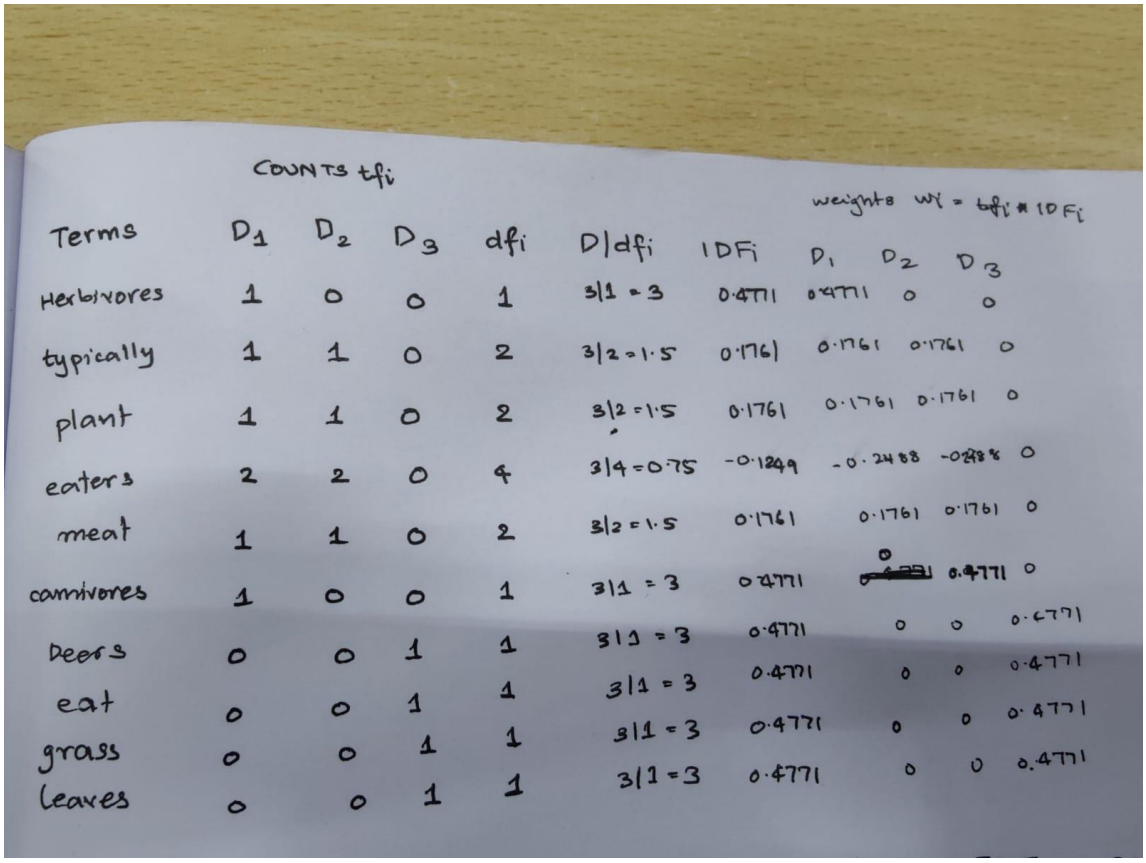
d_2: Carnivores are typically meat eaters and not plant eaters

d_3: Deers eat grass and leaves

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

Herbivores	d1
typically	d1, d2
plant	D1, d2
eaters	D1, d2
meat	D1, d2
carnivores	D1, d2
deers	d3
eat	d3
grass	d3
leaves	d3

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



COUNTS tfi

Terms	D_1	D_2	D_3	dfi	D/dfi	IDFi	weights W_i = tfi * IDFi D_1	D_2	D_3
Herbivores	1	0	0	1	3/1 = 3	0.4771	0.4771	0	0
typically	1	1	0	2	3/2 = 1.5	0.1761	0.1761	0.1761	0
plant	1	1	0	2	3/2 = 1.5	0.1761	0.1761	0.1761	0
eaters	2	2	0	4	3/4 = 0.75	-0.1249	-0.2488	-0.2488	0
meat	1	1	0	2	3/2 = 1.5	0.1761	0.1761	0.1761	0
carnivores	1	0	0	1	3/1 = 3	0.4771	0	0.4771	0
Deers	0	0	1	1	3/1 = 3	0.4771	0	0	0.4771
eat	0	0	1	1	3/1 = 3	0.4771	0	0	0.4771
grass	0	0	1	1	3/1 = 3	0.4771	0	0	0.4771
leaves	0	0	1	1	3/1 = 3	0.4771	0	0	0.4771

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

d1 and d2 would be retrieved based on the inverted index above.

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

Cosine Similarity calculations: **Similarity between query and document 1: 0.464916** **Similarity between query and document 2: 0.464916** **Similarity between query and document 3: 0** **Ranking documents: D1 = D2 > D3**

Is the ordering desirable? If no, why not?:
No, the ordering is not desirable as document 3 is also relevant to the query but has not been retrieved.

[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.

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

    Parameters
    ----------
    arg1 : list
        A list of lists of lists where each sub-list is
        a document and each sub-sub-list is a sentence of the document
    arg2 : list
        A list of integers denoting IDs of the documents
    Returns
    -------
    None
    """
    index = {tokens: [] for d in docs for sentence in d for tokens in sentence}
    # Iterate through each document
    for doc_id, doc in zip(docIDs, docs):
        # Iterate through each sentence in the document
        doc_t = [token for sent in doc for token in sent]
        for sentence in doc:
            # Update the document index with terms and their frequencies
            for term in sentence:
                if term in index.keys():
                    if (doc_id, doc_t.count(term)) not in index[term]:
                        index[term].append((doc_id, doc_t.count(term)))

    self.index = (index, len(docIDs), docIDs)
```

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

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


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

    doc_IDs_ordered = []
    index, doc_num , doc_ID = self.index

    Doc = np.zeros((doc_num, len(index.keys())))
    key = list(index.keys())

    for i in range(len(key)):
        for l in index[key[i]]:
            Doc[l[0]-1, i] = l[1]

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

    # constructing tf-idf matrix
    for i in range(tf.shape[0]):
        tf[i, :] = Doc[i, :] * idf.T

    for i in range(len(queries)):
        query = defaultdict(list)
        for j in queries[i]:
            for word in j:
                if word in index.keys():
                    query[word] = index[word]
```

```
    query = dict(query)
    Query = np.zeros((1, len(key)))
    for m in range(len(key)):
        if key[m] in query.keys():
            Query[0, m] = 1

    Query = Query*idf.T

    # cosine similarity
    ranks = []
    for d in range(tf.shape[0]):
        similarities = np.dot(Query[0, :], tf[d, :])/((np.linalg.norm(Query[0, :]) + 1e-4)*(np.linalg.norm(tf[d, :]) + 1e-4))
        ranks.append(similarities)
    doc_IDs_ordered .append([x for _, x in sorted(zip(ranks, doc_ID), reverse=True)])
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):
    """
    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
    # num_retrieved_docs = len(query_doc_IDs_ordered)
    # if num_retrieved_docs >= k:
    num_true_docs_retrieved = 0
    for i in range(k):
        if query_doc_IDs_ordered[i] in true_doc_IDs:
            num_true_docs_retrieved += 1
    precision = num_true_docs_retrieved/k
    return precision
```

Recall@k:

```
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
    # num_retrieved_docs = len(query_doc_IDs_ordered)
    num_true_docs = len(true_doc_IDs)
    num_true_docs_retrieved = 0
    # if num_retrieved_docs >= k:
    for i in range(k):
        if int(query_doc_IDs_ordered[i]) in true_doc_IDs:
            num_true_docs_retrieved += 1
    recall = num_true_docs_retrieved/num_true_docs
    return recall
```

$F_{0.5}$ score@k:

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

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

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

    fscore = -1

    #Fill in code here
    precision = self.queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    recall = self.queryRecall(query_doc_IDs_ordered, query_id, true_doc_IDs, k)
    fscore = statistics.harmonic_mean([precision, recall])
    return fscore
```

AP@k:

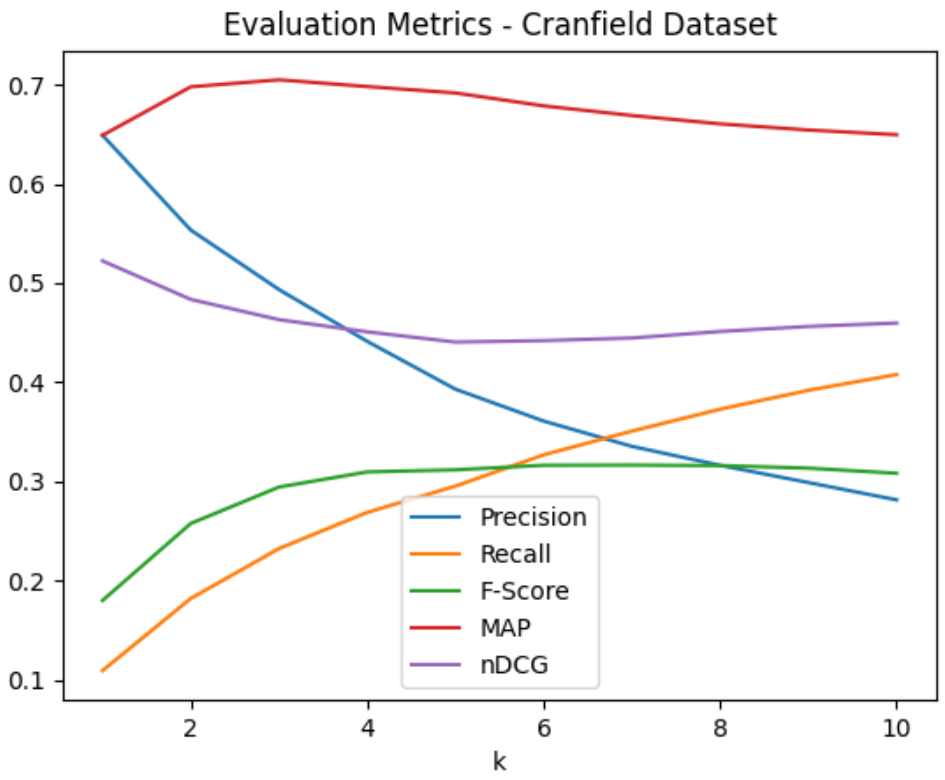
```
def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    """
    Computation of average precision of the Information Retrieval System
    at a given value of k for a single query (the average of precision@i
    values for i such that the ith document is truly relevant)

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

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

    #Fill in code here
    # num_retrieved_docs = len(query_doc_IDs_ordered)
    # if num_retrieved_docs >= k:
    precisions = []
    i = 0
    for i in range(k):
        if (query_doc_IDs_ordered[i] in true_doc_IDs):
            precision = self.queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, i+1)
            precisions.append(precision)
    if len(precisions) == 0:
        return 0
    avgPrecision = np.mean(precisions)
    return avgPrecision
```

nDCG@k:

```
def queryNDCG(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):

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

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

    relevance_vals = {}
    relevant_docs = []

    dcg = 0.0
    idcg = 0.0
    nDCG = 0

    for dict in qrels:
        for doc_id in true_doc_IDs:
            if int(dict['id']) == doc_id and int(dict['query_num']) == query_id:
                relevance_vals[doc_id] = 5 - int(dict['position'])
                relevant_docs.append(int(doc_id))

    for i in range(k):
        doc_id = int(query_doc_IDs_ordered[i])
        if doc_id in relevant_docs:
            relevance = relevance_vals[doc_id]
            dcg = dcg + (relevance/math.log2(i + 2))

    optimal_order = sorted(relevance_vals.values(), reverse = True)
    num_relevant_docs = len(optimal_order)

    num_docs_idcg = min(num_relevant_docs, k)
    for i in range(num_docs_idcg):
        relevance = optimal_order[i]
        idcg = idcg + (relevance/math.log2(i + 2))

    if idcg != 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:
Firstly, we observe that two functions seem to be monotonic from the graph – precision and recall. Precision decreases with an increase in the value of k in a non-linear fashion. Recall, by definition, either increases or stays constant, and it is observed that it is increasing with increase in the value of k. The F-score, which is a weighted harmonic mean of precision and recall, increases up to a certain point and constant after that. The nDCG graph shows that it starts high and then decreases, but may increase with an increase in the value of k as the graph seems to be rising/constant from k = 5. Since the ideal value of nDCG = 1, there is room for improvement. The MAP graph shows that it first increases and starts going down after k = 3.

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

16.41s

[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:

- Assuming that the words are orthogonal to each other, that they are not related to each other
- Not looking at the underlying semantic relationship between words, of the query and of the documents
- Returning irrelevant documents just because they show up in the query words and the document words
- High dimensionality, sparse representation of documents

Examples from your results:
(..) indicates that those docs were retrieved because they contained those words
{Query} - Location of halted movement for a non sharp object within a supersonic flow
{doc ID needed} - 35
{doc IDs obtained} - 228 (totally irrelevant), 1300 (bluntness, supersonic are mentioned), 401 (Hypersonic, airflow), 690(Flow, spiked cylinder)

{Query} - Some research on high speed flutter exploration into phenomenon of rapid self excited vibrations of a wing of an aircraft
{doc ID needed} - 1111
{doc IDs obtained} - 100 (vibration, aircraft), 908 (vibration), 51(aircraft), 345(aircraft), 844(vibrations)

{Query} - Constraints on structuring for cost effectiveness in missile frameworks
{doc ID needed} - 834
{doc IDs obtained} - 32(missile), 1217(constraints), 256(totally irrelevant), 834(missile)

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.