CS6370: Natural Language Processing
Project

Release Date: 21st March 2024 Deadline:

Name:	Roll No.:
Aswin Balamurugan	CH20B018
Tushar Bhutada	CH20B025
Ashwin K Krishna	CH20B017

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.

Term	Documents
herbivores	d1
typically	d1,d2
plant	d1,d2
eaters	d1,d2
meat	d1,d2
carnivores	d2
deer	d3
eat	d3
grass	d3
leaves	d3

The table shows the inverse document representation of the given sentences

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

Term	d1	d2	d3	N/n	IDF
herbivores	1	0	0	3	0.4771
typically	1	1	0	1.5	0.1761
plant	1	1	0	1.5	0.1761
eaters	2	2	0	1.5	0.1761
meat	1	1	0	1.5	0.1761
carnivores	0	1	0	3	0.4771
deer	0	0	1	3	0.4771
eat	0	0	1	3	0.4771

grass	0	0	1	3	0.4771
leaves	0	0	1	3	0.4771

This table shows the TF and the IDF values of the terms in the document
Here N = total number of documents (since there are three sentences, N=3)
n = number of docs with a particular type

Term	d1	d2	d3
herbivores	0.4771	0	0
typically	0.1761	0.1761	0
plant	0.1761	0.1761	0
eaters	0.3522	0.3522	0
meat	0.1761	0.1761	0
carnivores	0	0.4771	0
deer	0	0	0.4771
eat	0	0	0.4771
grass	0	0	0.4771
leaves	0	0	0.4771

This table shows the TF-IDF representation of the given documents
TF-IDF = TF x IDF

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

First “plant eaters” would be tokenized as “plant” and “eaters”. The term “plant” is in d1 and d2 and the term “eaters” is again in both d1 and d2. So the documents retrieved would be **d1** and **d2**.

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

Cosine Similarity calculations:
For calculating cosine similarity, the TF-IDF of the query needs to be calculated.

Term	Query (TF)	Query (TF-IDF)
herbivores	0	0
typically	0	0
plant	1	0.1761
eaters	1	0.1761
meat	0	0
carnivores	0	0
deer	0	0
eat	0	0
grass	0	0
leaves	0	0

For cosine similarity we perform dot product with each document

$$Cosine\ Similarity = \frac{dot\ product\ (Query, Doc)}{Norm(Query) * Norm(Doc)}$$

$$Cosine\ Similarity\ (Query, d1) = \frac{0.031 + 0.062}{0.249 * 0.668} = 0.559$$

$$Cosine\ Similarity\ (Query, d2) = \frac{0.062 + 0.031}{0.249 * 0.668} = 0.559$$

$$Cosine\ Similarity\ (Query, d3) = \frac{0}{0.249 * 0.9542} = 0$$

Ranking documents:
Both d1 and d2 have the same cosine similarity, hence the only way to order them is by rank in the document list which is d1 followed by d2. If we don't follow that then both should be ranked equally. The cosine similarity of d3 is zero and should be ranked last.

Is the ordering desirable? If no, why not?:
No, the IR system has ranked both d1 and d2 equally although d1 is more relevant than d2. Also, d3 turns out to be completely ignored despite having more relevance with the query than d2. Hence, this is not the best ranking. Ideally, it should have been d1, d3, and d2 in the given order.

[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.

```
from util import *

# Add your import statements here
import numpy as np
from textblob import TextBlob
from spellchecker import SpellChecker
from numpy.linalg import svd
import enchant
import sys
movies_dict = enchant.PyPWL("word.txt")

class InformationRetrieval():

    def __init__(self):
        self.index = None

    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
        """

        doc_inverted_list = {}
        for i in range(0,len(docs)):
            for j in range(0,len(docs[i])):
                for k in range(0,len(docs[i][j])):
                    if docs[i][j][k] not in doc_inverted_list:
                        doc_inverted_list[docs[i][j][k]] = [docIDs[i]]
                    elif docIDs[i] not in doc_inverted_list[docs[i][j][k]] :
```

```
                        doc_inverted_list[docs[i][j][k]].append(docIDs[i])

        self.index = (doc_inverted_list,docIDs,docs)


    def rank(self, queries, K = 400, w = 1.0):
        """
        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 = []

        # Access elements from the index (assuming appropriate structure)
        doc_inverted_list, doc_no, docs = self.index
        n_docs = len(docs)
        Query = queries

        # Spell correction using TextBlob and movies_dict
        spell = SpellChecker()
        for query_i in range(0,len(Query)):
            for sent_j in range(0,len(Query[query_i])):
                # Check for misspelled words and correct if necessary
                misspelled = spell.unknown(Query[query_i][sent_j])
                for word_k in range(0,len(Query[query_i][sent_j])):
                    if Query[query_i][sent_j][word_k] not in
doc_inverted_list:

                        b = TextBlob(Query[query_i][sent_j][word_k])
                        b1 = Query[query_i][sent_j][word_k]
                        Query[query_i][sent_j][word_k] = str(b.correct())
                        b = str(b.correct())
                        if b == b1 and b in misspelled:
```

```
                            suggestions = movies_dict.suggest(b1)
                            if len(suggestions) == 0:
                                Query[query_i][sent_j][word_k] = b1 # Keep
original word if no suggestions
                            else:
                                b = suggestions[0]
                                Query[query_i][sent_j][word_k] =
suggestions[0] # Use first suggestion
                    # Handle words not in the inverted list and add them with
empty document list
                    if Query[query_i][sent_j][word_k] not in doc_inverted_list
:
                        doc_inverted_list[Query[query_i][sent_j][word_k]] =
[0]

        # Doc-term matrix and word types initialization
        doc_term_matrix = np.zeros((n_docs, len(doc_inverted_list.keys())))
        words = list(doc_inverted_list.keys())

        # IDF calculation for all words
        idf = np.zeros((len(words), 1))
        for i in range(len(words)):
            idf[i] = np.log2(n_docs / (len(doc_inverted_list[words[i]])))

        # Find term frequencies and fill doc-term matrix
        for doc_i in range(len(docs)):
            for sent_j in range(len(docs[doc_i])):
                for word_k in range(0, len(docs[doc_i][sent_j])):
                    word = docs[doc_i][sent_j][word_k]
                    if word in doc_inverted_list:
                        ind = words.index(word)

                        # When not using weights
                        doc_term_matrix[doc_i][ind] += 1

                        # Below part for using weights
                        if sent_j == len(docs[doc_i])-1:
                            doc_term_matrix[doc_i][ind] += w
                        else:
                            doc_term_matrix[doc_i][ind] += 1

        # Convert TF to TF-IDF in doc-term matrix
        for doc_vec in range(doc_term_matrix.shape[0]):
            doc_term_matrix[doc_vec, :] = doc_term_matrix[doc_vec, :] * idf.T

        # Get TF values for query-term matrix
```

```
        query_term_matrix = np.zeros((len(Query),
len(doc_inverted_list.keys())))
        for query_i in range(0, len(Query)):
            for sent_j in range(0, len(Query[query_i])):
                for word_k in range(0, len(Query[query_i][sent_j])):
                    word = Query[query_i][sent_j][word_k]
                    if word in doc_inverted_list:
                        ind = list(doc_inverted_list.keys()).index(word)
                        query_term_matrix[query_i][ind] += 1

        # Convert TF to TF-IDF in query-term matrix
        for doc_index in range(query_term_matrix.shape[0]):
            query_term_matrix[doc_index, :] = query_term_matrix[doc_index, :]
* idf.T

        # Perform LSA
        U, S, V_T = svd(doc_term_matrix.T, full_matrices=False)

        Uk = U[:, :K]
        Sk = np.diag(S[:K])
        Vk_T = V_T[:K]

        # Update doc_term_matrix and query_term_matrix using LSA components
        doc_term_matrix = Vk_T.T @ Sk
        query_term_matrix = query_term_matrix @ Uk

        for query_i in range(len(query_term_matrix)):

            temp = []
            for word_vec_j in range(len(doc_term_matrix)):
                # Calculate cosine similarity between query vector and
document vector
                cossim = np.dot(query_term_matrix[query_i, :],
doc_term_matrix[word_vec_j, :]) / \
                        ((np.linalg.norm(query_term_matrix[query_i, :]) + 1e-
6) *  # Add epsilon for numerical stability
                        (np.linalg.norm(doc_term_matrix[word_vec_j, :]) + 1e-
6))

                temp.append(cossim)

            # Sort document IDs by cosine similarity (descending order)
            sorted_doc_cosines = sorted(zip(temp, doc_no), reverse=True)

            # Extract document IDs from sorted cosine similarities
            doc_IDs_ordered.append([x for _, x in sorted_doc_cosines])
```

```
        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
    """

    # Calculate the intersection size between retrieved and relevant documents at rank k
    intersection_size = len(set(query_doc_IDs_ordered[:k]) & set(true_doc_IDs))

    # Calculate precision at rank k
    precision = intersection_size / k

    return precision
```

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
    """

    # Calculate the intersection size between retrieved and relevant documents at rank k
    intersection_size = len(set(query_doc_IDs_ordered[:k]) & set(true_doc_IDs))

    # Calculate recall at rank k, 0 if len(true_doc_IDs) <= 0
    recall = intersection_size / len(true_doc_IDs) if len(true_doc_IDs) > 0 else 0

    return recall
```

$F_{0.5}$ 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
    """

    # Calculate precision and recall for the current query
    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)

    # Handle division by zero for F-score calculation
    if precision == 0 and recall == 0:
        fscore = 0
    else:
        # Calculate F1-score (harmonic mean of precision and recall)
        fscore = (2 * precision * recall) / (precision + recall)

    return fscore
```

AP@k:

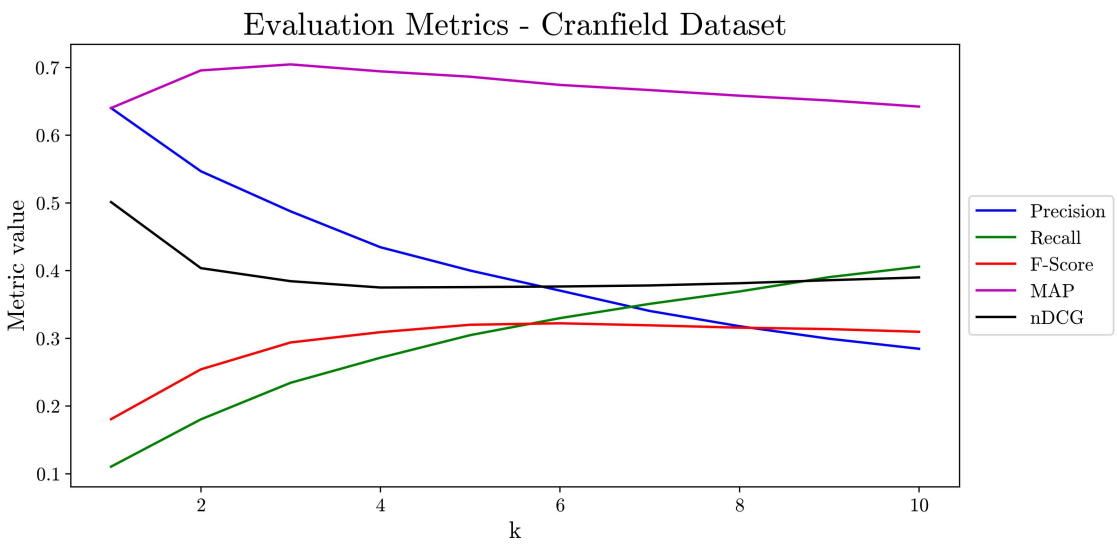
```python
def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
    """
    Computation of average precision of the Information Retrieval System
    at a given value of k for a single query (the average of precision@i
    values for i such that the ith document is truly relevant)

    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
    """

    count = 0  # Count of relevant documents retrieved
    p_sum = 0  # Sum of precisions at different retrieval positions

    for i in range(k):
        if query_doc_IDs_ordered[i] in  true_doc_IDs:  # Check if retrieved document ID exists in true doc IDs
            count += 1

            # Calculate precision at position (i + 1) using your defined queryPrecision function
            precision_at_i = self.queryPrecision(query_doc_IDs_ordered, query_id, true_doc_IDs, i + 1)
            p_sum += precision_at_i

    # Avoid division by zero
    avgPrecision = p_sum / count if count != 0 else 0

    return avgPrecision
```

nDCG@k:

```
# Relevance scores (initialization with zeros)
rel = np.zeros((len(query_doc_IDs_ordered), 1))

true_doc_IDs["position"] = 5-true_doc_IDs["position"]

# Sort documents by decreasing position (relevance or rank)
true_doc_ids_sorted = true_doc_IDs.sort_values("position", ascending=False)

# Ideal DCG (discounted sum of ideal positions)
iDCG = true_doc_ids_sorted.iloc[0]["position"]
for i in range(1, min(k,len(true_doc_IDs))):
    iDCG += true_doc_ids_sorted.iloc[i]["position"] / np.log2(i+1)

# List of document IDs from the true doc IDs subset (for efficiency)
t_docs = list(map(int, true_doc_IDs["id"]))

# Calculate relevance scores for retrieved documents
for i in range(k):
    if query_doc_IDs_ordered[i] in t_docs:
        rel[i] = true_doc_IDs[true_doc_IDs["id"] == str(query_doc_IDs_ordered[i])].iloc[0]["position"]

# Discounted Cumulative Gain (DCG)
DCG = 0
for i in range(k):
    DCG += rel[i] / np.log2(i + 2)

# Normalized Discounted Cumulative Gain (nDCG)
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:

- **Precision** decreases throughout the increase in k. This is expected as the fraction of relevant retrievals out of k total retrieved documents is supposed to decrease as k increases(except for some discontinuities at retrieval of relevant documents). But, as the MAP takes mean over all queries, those spikes get flattened and a gradually decreasing curve is obtained.
- **Recall** is increasing monotonically with k as the number of retrieved documents(k, denominator) is constant but the number of retrievals can only increase with the number of documents.
- **F-score** increases initially with k as the chances of relevant retrievals increase with the increase in k. But on further increase, it tends to flatten out. F-score is used to deal with the precision-recall trade-off. Flattening of the F-score curve typically indicates that the model's performance is reaching at an optimum level with the correct balance between precision and recall. Hence, we can see that around k = 6 or 7 we get this optimum level.
- **MAP**, initially, increases monotonically with k as the number of documents retrieved increases, the system is more likely to retrieve relevant documents, resulting in a higher average precision score and therefore a higher MAP. Then slightly decrease after k = 6.
- **nDCG** follows a similar pattern as that of F-score.

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

The time taken for IR system is approximately 16 seconds.

[Warm up] Part 4: Analysis [Theory]

1. What are the limitations of such a Vector space model? Provide examples from the cranfield dataset that illustrate these shortcomings in your IR system.

Limitations:

- As usually there are very large number of words in a corpus, the vector space and hence computations become too big and expensive.
- Everytime query or a document with new words is added, and most of the computations like inverted indexing, IDF calculation need to be repeated.
- The vector space approach is not able to handle **polysemy**. It retrieves irrelevant documents which have similar words as query but used in different context.
- It can not handle **synonymy** too. Documents with similar(not the same) words should be retrieved in a good IR engine, but this is not the case with the vector space approach.
- It does not consider the order of the words. If the **n-gram** vector space approach is used to resolve this issue, the computations become too expensive.

Examples from your results:

One of the areas where it fails is when related words are used and not the exact words. For example, **query 35** in the Cranfield data set is about acoustic wave propagation but nowhere in the relevant documents the words acoustic or wave or propagation is mentioned instead words like sound, pulse, and transmission are mentioned which are very closely related to the given query words.

The same applies if we use different word forms as well. For example, in query 28 the word linear is used but in its relevant documents the word linearized is mentioned and due to improper lemmatization by the nltk library, the query fails to fetch the relevant document.

Another area where it fails is when symbols are included(excluded) along with the words. For example in query 9 slip flow is written as /slip flow/, although there are documents relevant to slip flow, due to improper processing by the ptb tokenizer the "/" is not removed and hence fails to fetch the relevant documents. Basically, it fails whenever the exact words including the word forms in the relevant documents are not used.

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 A_1 is better than A_2 with respect to the evaluation measure E in task T on a specific domain D under certain assumptions 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.