CS6370: Natural Language Processing
Project

Release Date: 21 March 2024 Deadline: 15 May 2024

Name:	Roll No.:
Rahul	EE23M079
Rishabh Singh Gaharwar	CS21B067
Adepu Vaishnavi	CS21B003

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.

The implementation for this part of the project is available in 'Project_Part_1.ipynb' file.The inverted index representation for the above documents is -

```
[('carnivor', [2]), ('deer', [3]), ('eat', [3]), ('eater',
[1, 2]), ('grass', [3]), ('herbivor', [1]), ('leav', [3]),
('meat', [1, 2]), ('plant', [1, 2]), ('typic', [1, 2])]
[('carnivor', [2]), ('deer', [3]), ('eat', [3]), ('eater',
[1, 2]), ('grass', [3]), ('herbivor', [1]), ('leav', [3]),
('meat', [1, 2]),('plant', [1, 2]), ('typic', [1, 2])]
```

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

The TF_IDF term-doc matrix is as below-

Term	Doc 1	Doc 2	Doc 3
'carnivor'	0.0	0.4771	0.0
'deer'	0.0	0.0	0.4771
'eat'	0.0	0.0	0.4771
'eater'	0.3521	0.3521	0.0
'grass'	0.0	0.0	0.4771
'herbivor'	0.4771	0.0	0.0

'leav'	0.0	0.0	0.4771
'meat'	0.1761	0.1761	0.0
'plant'	0.1761	0.1761	0.0
'typic'	0.1761	0.1761	0.0

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

The documents retrieved are **doc1,doc2**. These two documents contain both terms of the query. Cosine similarity values also confirm this.

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

Cosine Similarity calculations:

Document	**Cosine Similarity Value**
doc1	0.56015
doc2	0.56015
doc3	0.0

Ranking documents: Ranking done based on the similarity value will assign the same rank to both document 1 and 2.

Doc 1 = Doc 2 > Doc 3

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

This ordering is **not** desirable as this method clearly states that there is zero similarity between document 3 and the query. But we do know ‘plant

eaters' is related to document 3 from human judgment. Further between document 1 and document 2 humans would usually rank document 1 higher than document 2 as the first one talks about the plant eaters (herbivores). This isn't captured in the above method.

[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 first part of IR is **buildIndex** with which we capture all the information we need to know about the documents. In this we store the term frequency of each term present in the corpus-

The code is as below -

```
index = None
# Initialize index as a nested dictionary with types as keys and empty dictionaries as values
index={tokens: {} for d in docs for sentence in d for tokens in sentence}
# List to flatten the documents into a single list of tokens
flattened_docs=[]

# Iterate through each document, sentence, and token to build the flattened_docs list
for doc in docs:
    flattened_docs.append([token for sentence in doc for token in sentence])

# Iterate through each document, sentence, and term to update the index dictionary
for doc_ind in range(len(docs)):
    for sentence in docs[doc_ind]:
        for term in sentence:
            # Check if the document ID is not already in the index for the current term
            if docIDs[doc_ind] not in index[term]:
                # Count the frequency of the term in the flattened document and update the index
                index[term][docIDs[doc_ind]]=flattened_docs[doc_ind].count(term)

# Store the index and document IDs in the class attributes
self.index = index
self.doc_IDs = docIDs
```

The next part of the IR system is the **rank** function, which using the extracted information from buildIndex ranks the **1400** documents present based on their relevance for a query.

We build the tf-idf matrix first similar to the one done in part1 -

```
doc_IDs_ordered = []
# Retrieve index and document IDs from class attributes
index=self.index
docIDs=self.doc_IDs

# Create a list of unique terms and get the counts of documents and terms
term_list=list(index.keys())
doc_count=len(docIDs)
term_count=len(term_list)

# Calculate inverse document frequency (IDF) for each term and store them
#in a dictionary with unique terms as keys
idf={}
for term in term_list:
    idf[term]=log10(len(docIDs)/len(index[term]))

# Initialize a matrix to store the tf-idf values
doc_mat=np.zeros((term_count,doc_count))

# Populate the term-document matrix with weighted term frequencies
for i in range(term_count):
    for j in range(doc_count):
        # Check if the current document ID exists in the index for the current term
        #if not present tf-idf=0 for that document ID , current term
        if j+1 in index[term_list[i]]:
            # Multiply the term frequency by the inverse document frequency (IDF)
            doc_mat[i][j]= index[term_list[i]][j+1] * idf[term_list[i]]
# Transpose the matrix for further processing
doc_mat_transpose=doc_mat.T
```

Using the above matrix and the IDF dictionary we will create vector representations for the queries like below and rank documents -

```
for query in queries:
    # Flatten the query into a list of tokens
    flattened_query=[token for sentence in query for token in sentence]

    # Initialize a vector to represent the query
    query_vec=np.zeros(term_count)

    # Calculate the tf*idf values for each term for the query vector
    for i in range(term_count):
        query_vec[i]=flattened_query.count(term_list[i])*idf[term_list[i]]

    # Initialize a dictionary to store document similarities
    # With keys as documentIDs and values are cosine sim values
    sim_dict={}

    # Calculate cosine sim of the query with each document and store it in the above dict
    vector2 = np.array(query_vec)
    magnitude2 = np.linalg.norm(vector2)
    for doc_id in range(doc_count):
        vector1 = np.array(doc_mat_transpose[doc_id])
        dot_product = np.dot(vector1, vector2)
        magnitude1 = np.linalg.norm(vector1)
        if dot_product>0:
            sim_dict[doc_id+1]=dot_product / (magnitude1 * magnitude2)
        else:
            sim_dict[doc_id+1]=0

    # Sort document similarities in descending order
    sorted_tuples = sorted(sim_dict.items(), key=lambda item: item[1], reverse=True)
    # Obatin the keys (i.e documentIDs with similarities in descending order)
    sorted_keys = [key for key, value in sorted_tuples]

    # Append ordered document IDs to the list, done for each query
    doc_IDs_ordered.append(sorted_keys)

# Return the list of ordered document IDs for each query
return doc IDs ordered
```

This completes VSM IR.

[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: Precision at k refers to the ratio of relevant documents in the first k documents obtained after ranking by k

$$Precision@k = \frac{\#\ of\ relevant\ documents\ in\ the\ first\ k\ documents}{k}$$

Implementation of precision@k is as follows :

```
def queryPrecision(self, query_doc_IDs_ordered, query_id,
true_doc_IDs, k):
        precision = 0
        # If there are no retrieved documents, return 0 precision
        if(len(query_doc_IDs_ordered)==0):
            return precision
        # Count relevant documents for the given query_id
        relevantDocs = []
        for entry in true_doc_IDs:
            if int(query_id)==int(entry['query_num']):
                relevantDocs.append(int(entry['id']))
        # Find number of relevant documents in retrieved documents
        for i in range(min(k, len(query_doc_IDs_ordered))):
            docID = query_doc_IDs_ordered[i]
            if int(docID) in relevantDocs:
                precision += 1
          # Calculate precision as fraction of relevant documents
retrieved
        precision /= k
        return precision
```

Recall@k: Recall at k refers to the ratio of relevant documents in the first k documents obtained after ranking by the total number of documents.

$$Recall@k = \frac{\#\ of\ relevant\ documents\ in\ the\ first\ k\ documents}{Total\ number\ of\ relevant\ documents}$$

Implementation of recall@k is as follows:

```
def queryRecall(self, query_doc_IDs_ordered, query_id,
true_doc_IDs, k):
        recall = 0
        # If there are no retrieved documents, return 0 recall
        if(len(query_doc_IDs_ordered)==0):
            return recall
        # Count total relevant documents for the given query_id
        total_relevant = 0
        relevantDocs = []
        for entry in true_doc_IDs:
            if int(query_id)==int(entry['query_num']):
                relevantDocs.append(int(entry['id']))
```

```
        total_relevant = len(relevantDocs)
        # If there are no relevant documents for the given
query_id, return 0 recall
        if total_relevant==0:
            return recall
        recall = 0
        # Find number of relevant documents
        for i in range(min(k, len(query_doc_IDs_ordered))):
            docID = query_doc_IDs_ordered[i]
            if int(docID) in relevantDocs:
                recall += 1
        # Calculate recall as the fraction of relevant documents
retrieved
        recall /= total_relevant
        return recall
```

$F_{0.5}$ score@k: F_β score at k is given as a function of Precision@k and Recall@k. The score is given by :

$$\frac{(\beta^2+1)Precision@k \times Recall@k}{\beta^2 \times Precision@k + Recall@k}$$

Implementation of F_β is as follows:

```
def queryFscore(self, query_doc_IDs_ordered, query_id,
true_doc_IDs, k, beta=1):
        fscore = -1
        # Calculate recall and precision
         recall = self.queryRecall(query_doc_IDs_ordered, query_id,
true_doc_IDs, k)
            precision = self.queryPrecision(query_doc_IDs_ordered,
query_id, true_doc_IDs, k)
        # If either recall or precision is 0, return 0 fscore
        if recall==0 or precision==0:
            return 0
        # Find fscore using the specified beta value
        fscore = (beta*beta + 1)*precision*recall
        fscore /= ((beta*beta)*precision + recall)
        return fscore
```

AP@k: Average precision at k is the value calculated by averaging the precision values obtained at each relevant document's rank up to k. Implementation of Average Precision is as follows:

```
def queryAveragePrecision(self, query_doc_IDs_ordered, query_id, true_doc_IDs, k):
        avgPrecision = 0
          # If there are no retrieved documents, return 0 average precision
        if(len(query_doc_IDs_ordered)==0):
            return avgPrecision
        # Count relevant documents for the given query_id
        relevantDocCount = 0
        relevantDocs = []
        for entry in true_doc_IDs:
            if int(query_id)==int(entry['query_num']):
                relevantDocs.append(int(entry['id']))
        # Calculate average precision
        for i in range(min(k, len(query_doc_IDs_ordered))):
            docID = int(query_doc_IDs_ordered[i])
            if docID in relevantDocs:
                relevantDocCount += 1
                avgPrecision += relevantDocCount/(i+1)
          # If there are no relevant documents, return 0 average precision
        if relevantDocCount==0:
            return 0
        # Calculate average precision
        avgPrecision /= relevantDocCount
        return avgPrecision
```

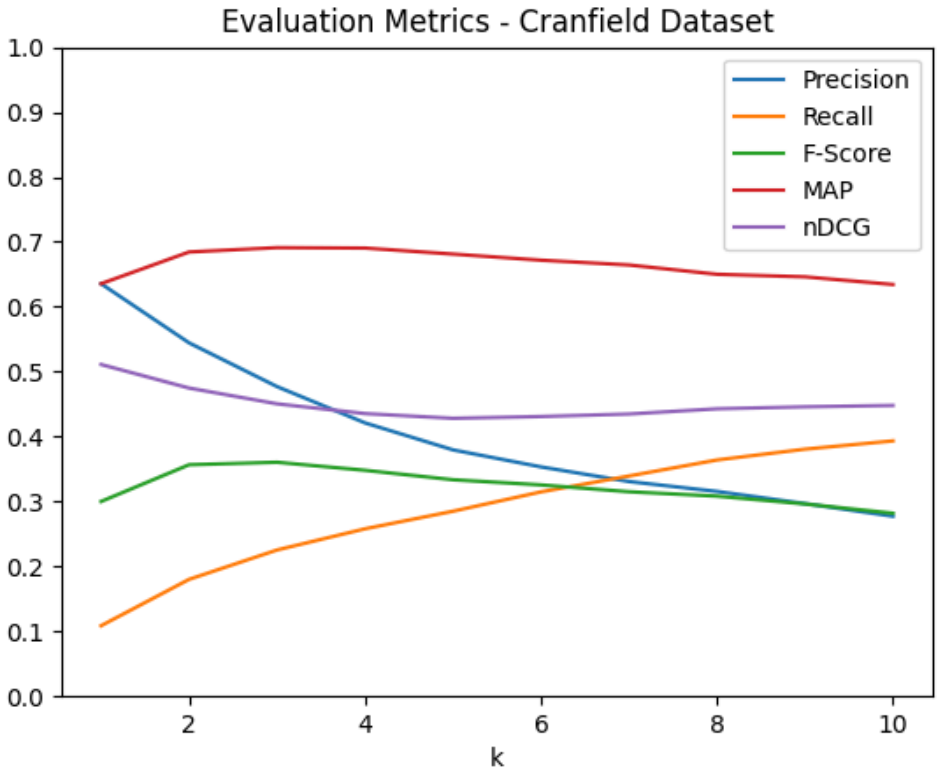
nDCG@k: nDCG stands for normalized discounted cumulative gain. The discounted gain obtained from a document ranked i with relevance score rel is given by :

$$\frac{rel}{\log_2(i+1)}$$

Discounted cumulative gain obtained from k documents is the sum of discounted gain obtained from each document.

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: We obtained the below plot for our TF-IDF IR-



Observation: The behavior of precision and recall aligns with what was discussed in class: as the number of retrieved documents increases, precision tends to decrease while recall increases. Such sensitivity to the number of retrieved documents is undesirable, as it hinders making sound conclusions about the performance of different IR methods. It's noteworthy that the MAP value doesn't vary much with the number of retrieved documents, which is expected since it is a more comprehensive measure than precision alone. Similarly, for the F-score, although it should be noted

that these methods don't take the relevance score mentioned in the dataset into account. On the other hand, the nDCG measure considers relevance scores, making it a richer measure among all, and it doesn't vary much with the number of retrieved documents.

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

Elapsed time (excluding the pre-processing time): 19.223148822784424 seconds
To measure the above time we've made the below changes to the main.py-

```
# Build document index
#measure the start time
start_time = time.time()
self.informationRetriever.buildIndex(processedDocs, doc_ids)
# Rank the documents for each query
doc_IDs_ordered = self.informationRetriever.rank(processedQueries)
end_time = time.time()
#measure the end time
print("Elapsed time:",end_time - start_time , "seconds")
```

We've decided to report the actual time taken for the retrieval excluding the time taken to preprocess the documents and queries.

Time taken for the entire IR process including the pre-processing time is-

Elapsed time: 28.993536949157715 seconds

The pre-processing takes approximately 10 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:

1. **Bag of Words Model -** VSM treats each document as a bag of words, disregarding the order of words and their relationships within

the document. Also, the bag of words model assumes independence between each pair of words, which is not a valid assumption.

2. **Synonym -** In VSM, synonyms pose a significant drawback because they are treated as separate terms, leading to potential issues in capturing the semantic similarity between words.Hence, each unique term in the document collection forms a dimension in the vector space, two similar but distinct words will end up having orthogonal components.
3. **Polysemy -** In VSM, polysemous words are represented as a single term despite having different meanings.
4. **Dimensionality -** Increase in dimensions leads to sparsity in data, and since measuring cosine similarity is at the heart of VSM, due to high dimensionality similarity measure won't be accurate. In VSM, the number of dimensions is the total number of unique tokens present which is huge.

Examples from your results:

1. Query - “result on supersonic or hypersonic blunt body problem”
    a. Desired document - 93
    b. Retrieved documents:
        i. 1188 (“blunt”, “body” are the words present)
        ii. 211 (“supersonic”, “hypersonic”, “blunt”, “body” are the words present)
        iii. 116 (irrelevant to the query. “resultant” is present)
        iv. 278 (“supersonic” is the word present)
        v. 626 (“result”, “supersonic”, “hypersonic”, “blunted”, “body” are the words present)
    c. Cause of failure - Bag of Words assumption : Word order is not captured
2. Query - “buckling of cones under pressure”
    a. Desired document - 932
    b. Retrieved documents:
        i. 48 (“cone” is the word present)
        ii. 423 (“cone” and “pressure are the words present)
        iii. 225 (“cone” and “pressure are the words present)
        iv. 1303 (“cone” and “pressure are the words present)
        v. 58 (“cone” and “pressure are the words present)
    c. Cause of failure - Synonymy : Pressure and compression are synonyms, but it is not captured

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.