CS6200

Information Retrieval

Semester – Fall'18

Prof. Nada Naji

PROJECT MEMBERS

Dipanjan Haldar: Computer Science

Monisha Carol Karise: Computer Science

Sanhita Murthy: Computer Science

Table of Contents

1. INTRODUCTION:

The aim of the project is to implement various retrieval systems using IR concepts covered in lectures and programming assignments. The project consists of components which were implemented as a part of our previous programming assignments consisting of the document parsers, indexer, and scoring algorithms coupled with evaluation mechanisms.

For the scope of this project we cover the BM25, tf-idf, Smoothed Query Likelihood Model (Jelinek Mercer) and Lucene as our retrieval models and return the top 100 documents for CACM test-collection(corpus). We implemented text processing including stemming and stopping which were also performed to compare different retrieval systems.

We have modified the implementations of standard algorithms like Luhn's algorithm for Snippet Generation and Boolean Retrieval and the k-word proximity search algorithm for the Extra Credit problems implementation

To assess the effectiveness of each of these retrieval system runs evaluation techniques of Precision and Recall Mean Average Precision, Mean Reciprocal Rank and Precision at Values at K were used.

PROJECT OVERVIEW:

The project was divided into the following phases:

PHASE 1: Indexing and Retrieval

- Task 1: Building retrieval system with Lucene, BM25, tf-idf and Query Likelihood Model
- Task 2: Performing query enrichment on model run
- Task 3a: Performing stopping on the corpus and running all the retrieval system models.
- Task 3b: Running all the retrieval system models on the stemmed corpus and analyzing the results of the 3 interesting queries.

PHASE 2: Snippet generation and document highlighting

Implemented a *snippet generation* technique and *query term highlighting* with the results from BM25 run

PHASE 3: Evaluation Precision and Recall and plotting a Recall and Precision curve.

In this phase we evaluated the different retrieval systems in term of their effectiveness

Extra Credit: Advance Search

Design and implementation of an “advanced search technique that allows users to choose certain search criteria such as searching an exact phrase or finding results that contain *any* of the query terms. The documents are ranked by relevance

2. MEMBER CONTRIBUTION:

Phase 1:

Task 1 - Sanhitha Murthy

Task2 - Sanhitha Murthy

Task 3 - Monisha Carol Karise

Phase 2: Dipanjan Haldar

Phase 3: Monisha Carol Karise

Extra credit: Dipanjan Haldar

3. LITERATURE AND RESOURCES:

Pseudo relevance feedback:

Relevance feedback is an approach where the results from an initial query run is used to provide a result set. Relevance feedback requires user input. However, Pseudo relevance feedback is the automated version of this where the top k documents as a result of running the query for the first time is considered relevant.
In our approach we use snippet generation to get the most frequent terms to modify our query.

Snippet Generation:

Snippet is generated using the Luhn's algorithm given in the course text book (Search Engines, Information Retrieval in Practice). Luhn's approach was to rank each sentence in a document using a significance factor and to select the top sentences for the summary. The significance factor for a sentence is calculated based on the occurrence of significant words. Significant words are defined in his work as words of medium frequency in the document, where "medium" means that the frequency is between predefined high-frequency and low-frequency cutoff values.

We referred the following resources for the above implementation are can be found here

4. IMPLEMENTATION AND DISCUSSION:

PHASE 1: Indexing and Retrieval

Task 1- Build your own retrieval systems

Process:

- Parse the corpus and queries using document parsing techniques
- Create unigram inverted index of the corpus
- Calculate the BM25/TF-IDF/Smoothed Query Likelihood score for each document for each query. Return the top 100 highest ranked documents for each query.
- Print the ranked output to a file in the following format:
    - query_id Q0 document_id rank score system_name

BM25

BM25 extends the scoring function for the binary independence model to include document and query term weights. It is a bag-of-words retrieval that ranks the documents based on the query terms that appear in the given document and not on the relationship between the query terms within a document. The following formula is used to calculate the BM25 score:

$$\sum_{i\in Q} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n_i-R+r_i+0.5)} \cdot \frac{(k_1+1)f_i}{K+f_i} \cdot \frac{(k_2+1)qf_i}{k_2+qf_i}$$

ri – number of relevant documents that contain the term i.
ni – number of documents that contain the term i
N – total number of documents in the given collection
R – total number of relevant documents for the query
fi – frequency of the term i in the document
qfi – frequency of the term i in the query
k1 – value is taken as 1.2 as per TREC standards
k2 – value is taken as 1.2 as per TREC standards
K - k1((1-b) +b* dl/avdl) where dl is the document length and avdl is the average document length in the collection
b – value I taken as 0.75 as per TREC standards.

TF-IDF

TF-IDF is calculated by multiplying term frequency and inverse document frequency of each query term appearing in the document. The documents are then ranked by adding these calculated values.

- The term frequency is calculated using
  tf = log(1+fi)
  where fi is the frequency of the term i in the document that we are scoring.

- The inverse document frequency is calculated using
  idf = (N/(1+ni))
  where N is the total number of documents,
  ni is the number of documents that contain term i in them.

- tf-idf = tf *idf

- The tf-idf score increases with the number of occurrences within a document.
- The tf-idf score increases with the rarity of terms within a collection.

Smoothed Query Likelihood Model

The documents are ranked based on the Jelinek-Mercer smoothing technique of the query likelihood model. The documents are ranked by the probability that the query could be generated by the document model.

The formula of Jelinek-Mercer smoothing technique is:

$$\log P(Q|D) = \sum_{i=1}^{n} log\ ((1-\lambda)\ f_{qi,D}/\ |D| + \lambda\ c_{qi}/\ |C|)$$

λ = 0.35 as given in the specification
fqi, D - the frequency of the query term in the document.
|D| - the total number of words in the document
cqi i- the frequency of the query term in the collection
|C| is the total number of words in the collection

Lucene

Version 4.7.2 of Lucene is used as a default retrieval model. The clean corpus and the queries are passed to this model to tank the top 100 documents. Standard Analyzer is used.

Task 2: Query Enrichment

We have used BM25 ranking algorithm and pseudo relevance feedback for query expansion. In Pseudo relevance feedback, we consider the top k documents generated as relevant and doesn't need user interaction.

The steps used in our approach is:

- Initial query is run with BM25 as the retrieval model.
- Top 50 documents are considered relevant
- Snippet for these documents is created by taking significant factor into consideration. Luhn's law is used for snippet generation
- Calculate the most frequent terms in the snippet that is generated by the previous step
- Eliminate the stop words and pick the top 30 terms from the snippets
- The original query is modified by adding these terms to the query
- Finally rerun the modified query

We implemented the above process with the top 10,20,30 and 50 term. We finally chose 30 terms to calculate the pseudo relevance as the mean average precision was the highest for the top 30 terms. The same process was followed for the second step where we considered 10,20,30,50, 100 documents and top 50 gave us the best results.

Task 3: Stemming and Stopping

Task 3A- Stopping

Processing a text query should mirror the processing steps that are used for documents. Words in the query text must be transformed into the same terms that were produced by document texts, or there will be errors in the ranking. [MRS 6.2]

Hence in the implementation the stop words provided in common_words.txt are removed from the corpus as well as the query before performing indexing and running of the retrieval models.

Task 3b - Stemming:

The retrieval models generated were run on the stemmed version of the corpus and the stemmed queries. The following observations were made:

Query by Query analysis:

Query #1:

Non-stemmed query: portable operating system

Stemmed query: portabl oper system

The top documents of the stemmed BM25 run retrieved relevant documents which contain the terms 'portabl' , 'oper' and 'system' . However, the document at rank 1 in the stemmed run (CACM-3127) does not appear in the top ten documents retrieved in the non-stemmed BM25 run. There's a stark difference in the results in both the cases because This shows that stemming enhanced the results of BM25 to a great extent.

Query #2:

Non-stemmed: performance evaluation and modelling of computer systems

Stemmed: perform evalu and model of comput system

On observing the top ranked document in the Query Likelihood runs on this query, it is evident that stemming did not produce good results. The top ranked document in the non-stemmed run is a relevant document which contains the phrases "performance evaluation models" and "evaluating the performance of computer systems" and "modeling evaluation performance". However, the stemmed run could not retrieve this document as the top ranked document. This is because the stems of the query terms can give rise to a lot of non-relevant words in the context of the query topic. For example, perform is a stem for perform,performance,performing,performed,and performer and model is a stem for comput is a stem for compute,computer,computing,computation and computed. Hence due to the many instances of words that can be produced from these stems, any document which contain these stems would be considered relevant, even though it is not actually relevant.

Query #3:

Stemmed: appli stochast process

Non-stemmed: applied stochastic processes

The TFIDF runs on both stemmed and non-stemmed retrieve documents where occurrences of the term 'process' or 'processes' are higher. This is because TFIDF score is directly proportional to the term frequency of the various query terms. Hence the documents which have the highest term frequency will be ranked high regardless of other query terms. This is why the tf-idf scoring method did not work very well on both stemmed and non-stemmed versions for this query.

PHASE 2: Snippet Generation and Document Highlighting

We have used an approach which performs a variation of the Luhn's algorithm and returns a window of text matching the highest number of query words. The algorithm is implemented in the following steps:

- Open the raw html files from the cacm corpus and prettify them using Beautiful soup. We have decided on a window of W (40), the number of words to be displayed in snippet
- Get the ranking of the documents for a given query from the output of BM25 results executed on the cacm corpus
- Iterate over the query terms and stop words from the common_words.txt are excluded.
- For the remaining query terms, each sentence of the document is weighted based on the number of non-stopped query terms present in the sentence and check for the window W (40) which contains the greatest number of words from the query.
- To highlight, split the sentences into keywords, highlight the ones that are important and then combine them back into a sentence. We have utilized the dominate library to generate the html pages from python.
- The query terms that appear in the document title and snippet will be displayed in **"BOLD" except the terms which are stop-words.**

PHASE 3: Retrieval Models Evaluation

Evaluation Techniques:

Precision = |Relevant ∩ Retrieved| / |Retrieved|

Recall = |Relevant ∩ Retrieved| / |Relevant|

Mean Average Precision (MAP) is the mean arithmetic of the precision values over all the queries.

Reciprocal rank is defined as the reciprocal of the rank at which the first relevant document is retrieved.

Mean reciprocal rank (MRR) is the average of the reciprocal ranks over a set of queries

Interpolated precision recall curve: At each achieved level of recall we set the precision to be equal to the maximum precision at this or higher recall levels

The effectiveness of all the 8 runs were evaluated using MAP, MRR, P@K (K=5 and K=20) over all 64 queries. Queries that don't have any entries in the relevance judgment have been excluded from evaluation. Finally, an interpolated recall precision curve was plotted for query #10. The results are available here

Algorithm to plot the interpolated recall precision curve for a given query:

1. At a given recall value, it is possible to have different precision values. So, at each obtained recall level, we set precision to be maximum precision at that recall level. The resultant points are then plotted. This gives a saw-toothed shaped curve.
2. Now that we have the maximum precision at all recall levels, we calculate the maximum precision at that recall level or at a higher recall level. To do this, at each obtained recall level, we set the precision as the maximum precision at this level or a recall level observed at a higher rank.
3. Plot the recall precision values at standard recall intervals of [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7.8,0.9,1.0]

Extra Credit:

Exact match:

Definition - all query terms must appear and in the same order

Algorithm:

1. Create positional unigram index from the corpus (word: {doc1: [pos1, pos2], doc2: [pos10]})
2. Used an intersection technique to fetch all the documents from the corpus only if they contain all the query words. Created a list for all these retrieved relevant documents
3. For each relevant document, take the first query and pick the first relevant document from the relevant list and fetch all the position of the query terms in that document.
4. For each position entry of first query term, validate whether we can jump to any of the position of the next query term with an interval of 1+position of the earlier query term.
5. If it is possible add the document and the query to a final dictionary else discard the document and move to the next document and repeat step 4
6. Repeat steps 4 and 5 for all the queries
7. From step 5 we have a list of relevant documents satisfying the Exact match criteria for each query
8. Re-index the relevant documents for a query and perform BM25 to rank all the documents for the given query

Code snippet to calculate the exact match single term queries:

- result_list => the list of documents returned from step2
- dictionary => holds the positional unigram index created in step1
- final_rank_map=> dictionary containing the queryid and relevant documents for it after exact match

```
if len (query_terms) == 1:
   # print ("The query terms are" + str (query_terms))
   resultList = getPostingList (query_terms[0])
```

```
  if not resultList:
     print ("0 documents returned as there is no match for query no : " +str(query_id))
     return
  else:

     inverted_list = dictionary[query_terms[0]]
     doc_map = {}
     for entry in inverted_list.keys ():
        value = len (inverted_list[entry])
        for i in range (0, value):
           if query_id not in final_rank_map.keys ():
              final_rank_map[query_id] = [entry]
           else:
              val = final_rank_map[query_id]
              val.append (entry)
              final_rank_map[query_id] = val
```

Code snippet to calculate the exact match for multi-term queries:

- result_list => the list of documents returned from step2
- final_rank_map=> dictionary containing the queryid and relevant documents for it after exact match

```
# This is for exact-match query
def exact_match (resultList, query_terms, query_id):
     pos_doc_map = {}
  for document in resultList:
     pos_list = []
     count = 0
     for term in query_terms:
        if count < len (query_terms):
           position_list_term = dictionary[term][document]
           pos_list.append (position_list_term)
           count = count + 1
           # print ("The positions are :" + str (position_list_term))
     pos_doc_map[document] = pos_list
  # print ("The map is :" + str (pos_doc_map))
  for key in pos_doc_map.keys ():
     p_list = pos_doc_map[key]
     length = len (p_list)
           loopcounter = 0

     while loopcounter < len (p_list[0]):
        flag = 1
        value = p_list[0][loopcounter]
        i = 1
        if value + 1 not in p_list[i]:
           loopcounter = loopcounter + 1
        else:
            while i < length:
                if (i < length) and (value + 1 in p_list[i]):
                    value = value + 1
                    i = i + 1
                    # print (" The value is : " + str (value))
                else:
                    flag = 0
                    break
            loopcounter = loopcounter + 1

            if flag == 1:
                if query_id not in final_rank_map.keys ():
                    final_rank_map[query_id] = [key]
                else:
                    val = final_rank_map[query_id]
                    val.append (key)
                    final_rank_map[query_id] = val

    # print ("The final map is : " + str (final_rank_map))
```

Ordered exact match within proximity N search:

same as above but order matters and any two query terms should separate by no more than N other tokens.

Note: Though this question was not asked explicitly to be done, we thought of doing it while we were doing the Proximity based search for the Best match

Algorithm:

We followed the same approach like ExactMatch to come up with a list of documents where all query terms are presents and then modified the logic for finding the positions for each term within given N(we call that slider in our program) and then rank the final relevant list with BM25

Code snippet for exact_match_proximity(N):

checkinclusion (value, slider, p_list[i]) -> function to check for positions with the given slider

```
def checkinclusion (value, slider, pos_list):
    for entry in pos_list:
        if (entry > value) and (entry <= (value + slider)):
            return 1
    else:
        return 0
```

```
def exact_match_proximity (resultList, query_terms, query_id, slider):
    # print ("The resultList : " + str (resultList))
    # print ("The query terms are : " + str (query_terms))
```

```python
pos_doc_map = {}
for document in resultList:
    pos_list = []
    count = 0
    for term in query_terms:
        if count < len (query_terms):
            position_list_term = dictionary[term][document]
            pos_list.append (position_list_term)
            count = count + 1
            # print ("The positions are :" + str (position_list_term))
    pos_doc_map[document] = pos_list
#   print ("The map is :" + str (pos_doc_map))
for key in pos_doc_map.keys ():
    p_list = pos_doc_map[key]
    length = len (p_list)
    # print ("The p_list is : " + str (p_list))
    # print ("The length of p_list is : " + str (length))
    # print ("####### : " + str (p_list[0][0]))
    # print ("Length of p_list[0] = " + str (len (p_list[0])))
    loopcounter = 0

    while loopcounter < len (p_list[0]):
        flag = 1
        value = p_list[0][loopcounter]
        # print (" The value of element : " + str (value))
        i = 1
        checker = checkinclusion (value, slider, p_list[i])
        if checker == 0:
            loopcounter = loopcounter + 1
        else:
            while i < length:
                if (i < length) and checkinclusion (value, slider, p_list[i]) == 1:
                    value = value + slider
                    i = i + 1
                    # print (" The value is : " + str (value))
                else:
                    flag = 0
                    break
            loopcounter = loopcounter + 1
            if flag == 1:
                # finalList.append(key)
                # final_rank_map[query_id]=key
                if query_id not in final_rank_map.keys ():
                    final_rank_map[query_id] = [key]
                else:
                    val = final_rank_map[query_id]
                    val.append (key)
                    final_rank_map[query_id] = val
```

```
# print ("The final map is : " + str (final_rank_map))
```

Best match:

Definition: A document is shown in the results if it contains at least

Algorithm:

1. Create positional unigram index from the corpus (word: {doc1: [pos1, pos2], doc2: [pos10]})
2. Used a union technique to fetch all the documents from the corpus if they contain any one of the query words. Created a list for all these retrieved relevant documents
3. Re-index the relevant documents for a query and perform BM25 to rank all the documents for the given query

Code snippet for Best Match:

```
def getalldocuments (query):
    result = []
    for term in query:
        if getPostingList (term) is not None:
            result += getPostingList (term)
    return list (set (result))


def getPostingList (term):
    if term in dictionary.keys ():
        postingList = dictionary[term]
        # print("The term is : " + str(term) + " => and posting list is : " +str(postingList))
        keysList = []
        for keys in postingList:
            keysList.append (keys)
        keysList.sort ()
        # print ("The keysList is : " +str(keysList))
        return keysList
    else:
        return None
```

Ordered best match within proximity N search:

same as above but order matters and any two query terms should separate by no more than N other tokens.

Algorithm:

We modified the approach for BestMatch to come up with a list of documents any 2 query terms are separated by the slider or if any one of the terms is present. We implemented a logic for finding the positions for 2 positional terms within given N (we call that slider in our program) and then rank the final relevant list with BM25

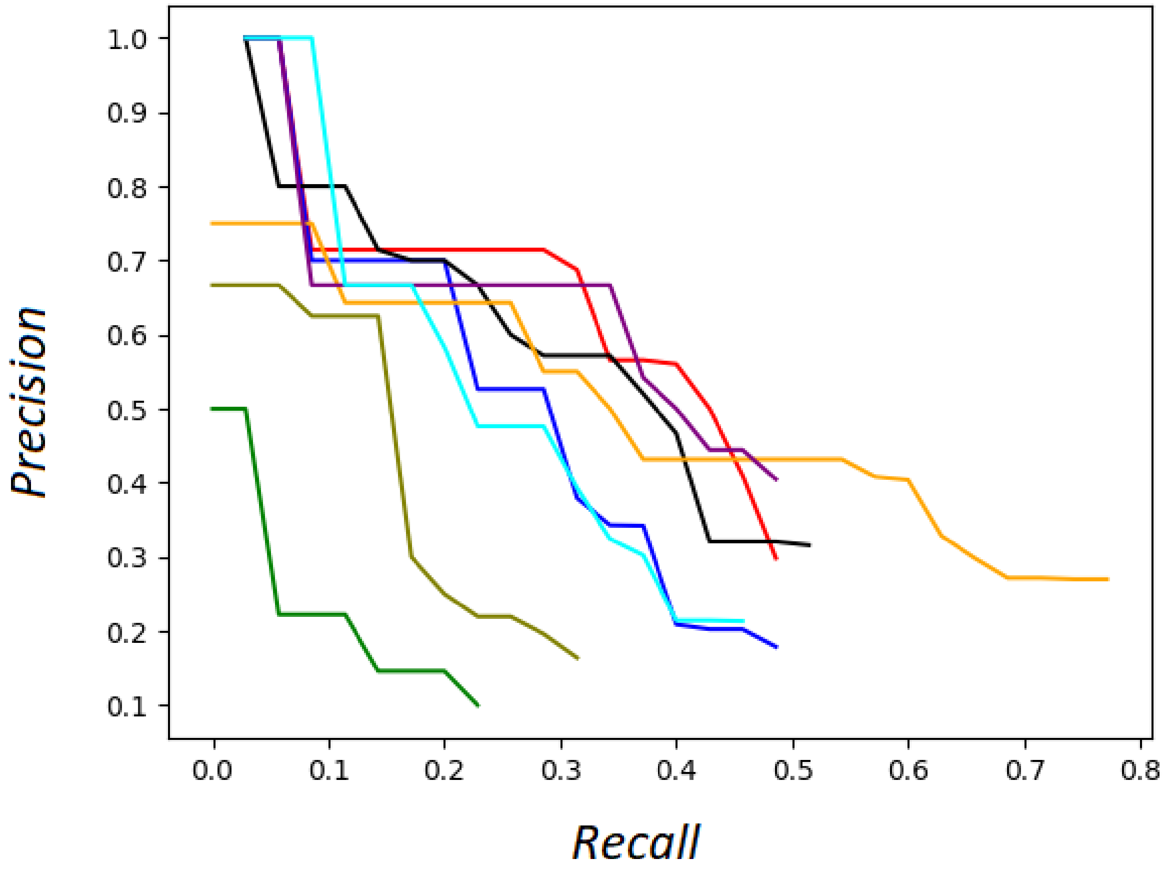
Note: The code and execution steps for each part is explained in the ReadMe.txt file inside the Extra credit directory

4. RESULTS:

The following results were obtained after using MAP and MRR techniques on the various runs in decreasing order of the MAP values:

Run	MRR	MAP
BM25 with Query Enrichment	0.5768	0.4315
BM25 (Stopping)	0.5117	0.4032
BM25 (No Stopping)	0.4875	0.3709
Lucene	0.4707	0.3535
Query Likelihood with JM Smoothing (No Stopping)	0.4338	0.3149
Query Likelihood with JM Smoothing (Stopping)	0.4743	0.2275
TF-IDF(Stopping)	0.3257	0.2138
TF-IDF (No Stopping)	0.1668	0.1358

The following is the **interpolated recall precision** curve for query number 10 (“parallel languages for parallel computation”)



🟩 TF-IDF (No stopping)

🟫 TF-IDF (Stopping)

🟦 Query Likelihood with JM Smoothing (No Stopping)

🟦 Query Likelihood with JM Smoothing (No Stopping)

🟥 BM25(No Stopping)

⬛ Lucene

🟪 BM25 (Stopping)

🟨 BM25 with Query Enrichment

5. CONCLUSION:

- BM-25 with Query Expansion yielded the best results and TF-IDF without stopping is the least effective scoring with respect to the MAP scores.
- Overall, all variations of BM-25 were extremely effective, followed by Lucene.
- Query Likelihood with JM Smoothing without stopping is almost as effective as Lucene. However, it did not perform well with stopping with a MAP score of 0.22 which is similar to the MAP score of TF-IDF with stopping.
- By observing the precision and recall values for any query we notice that as we go deeper in the list Recall increases. That is because recall increases every time, we see a relevant document. If we see a non-relevant document, then recall stays the same.
- The interpolated recall precision curve shows that precision decreases with recall. If we see a relevant document, the precision stays the same or goes higher. It can also happen that for a long-time precision is zero and then increases till the end of the list

6. BIBLIOGRAPHY:

- [CMS] Bruce Croft, Donald Metzler, Trevor Strohman, Search Engines: Information Retrieval in Practice, Publisher: Addison-Wesley, 2010.
- [MRS] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press. 2008.
- https://github.com/Knio/dominate
- http://www.cs.pomona.edu/~dkauchak/ir_project/whitepapers/Snippet-IL.pdf
- https://www.geeksforgeeks.org/luhn-algorithm/
- https://nlp.stanford.edu/IR-book/html/htmledition/relevance-feedback-and-query-expansion-1.html
- http://aclweb.org/anthology/C10-2007