# PHASE – 1:

Task 1A, Task 1B:

Implementation Process:

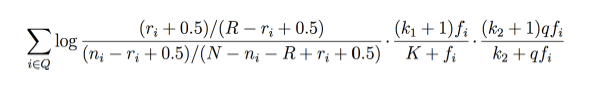
* Parse the corpus and the queries
* Create a unigram inverted index for the clean corpus
* Calculate the score (BM25/ TFIDF / Smoothed Query Likelihood) for each document for every query.
* Fetch the top 100 highest scored documents (top 100 ranks)
* Print the results in a file.

Format of the file:

*query\_id Q0 document\_id rank score system\_name*

1. 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:



ri – number of relevant documents that contain the term i.

ni – number of duments 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.

1. 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,

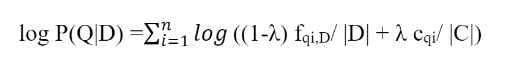
ni is the number of documents that contain term i in them.

* 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.

1. 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:



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

1. 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. StandardAnalyzer is used.

# Task 2:

Query Expansion:

Query expansion is a technique to reformulate the initial query for better performance of the search engine. This includes evaluation of user input and expansion of the given query to find the additional documents that are relevant to the given query. In our implementation, 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.

# PHASE – 2:

## Snippet Generation and Highlighting

The snippet generation incorporates concepts from Luhn’s algorithm mentioned in course textbook. The algorithm uses the concept of associating significance to sentences based on the query terms and the occurrence of significant words and ranking sentences based on significance. 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”.**

### References:

<https://github.com/Knio/dominate>

<http://www.cs.pomona.edu/~dkauchak/ir_project/whitepapers/Snippet-IL.pdf>

=

# Extra Credit:

## ***Exact match****: 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](#_Exact_match:_all) 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))  
 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***: 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

### Reference:

### <https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables>

### <https://github.com/itsnavneetk/booleanRetrieval>

**Note:** The code and execution steps for each part is explained in the ExtraCredit\_ReadMe.txt