**3. Implementation and Discussion:**

This section contains detailed description of the chosen models and design choices made during the implementation of these models. This section also includes query-by-query analysis of the run performed in the task3 of phase1.

**3.1 Task1- Phase1:**

**➢ BM25 Model:**

It’s an effective ranking algorithm based on binary independence model which also includes document and query term weights



-K1, K2 – parameters whose values are set empirically. We have takes K1 and K2 values as 1.2 and 100 respectively.



-dl= document length.

-b= 0.75, avdl= average document length

-ri= number of relevant documents containing the term i. We have taken ri=0

-R= number of relevant documents for the query. We have taken R=0

-ni= number of documents containing the term i.

-N= total number of documents in the collection.

-fi= frequency of the term i in the document

-qfi= frequency of the term i in the query

➢ Implementation:

• The list of documents is taken from the inverted index (unigram\_index.txt) for each term in the query.

• The score for each document is calculated using the above formula.

• The scores are stored in the dictionary and sorted in the descending order of their scores.

• Result containing the scores along with the document name is printed in the bm25\_Ranking.txt in the below format:

“query\_id Q0 doc\_id rank score system\_name”

➢ **Tf-idf Model:**

This is the simplest model which only considers term frequency and document frequency.

Score = (fi/dl) \* (1+ log (N/(ni+1)))

-dl= document length

-fi= frequency of the term i in the document

-N= total number of documents in the corpus

-ni= number of documents containing the term i

➢ Implementation:

• The list of documents is taken from the inverted index for each term in the query.

• A table containing unigram tokens with the name of the document is used to get document length for each of the document

• Score for each document for each term is calculated using the above-mentioned formula.

• The document and its respective score is stored as a dictionary and sorted based on the score in descending order.

• The result is printed into tfidf\_Ranking.txt in the below format:

“query\_id Q0 doc\_id rank score system\_name”

➢ **Smoothed Query Likelihood model:**

In this model documents are ranked by the probability that the query text could be generated by the language model. Query generation is the measure of how likely it is that a document is about the same topic as the query. Smoothing is included in this to avoid various estimation problem and to overcome data sparsity.



-fqi, D= number of times the term i occurs in the document D

-|D|= length of the document

-lambda= 0.35 as given in the problem statement

-Cqi= number of times the term i occurs in the whole collection of documents

-|C|= total length of all the documents present in the corpus.

➢ Implementation:

• The list of documents for each term of the query is taken from the inverted index.

• fqi is calculated for each term with the help of inverted index generated in unigram\_index.txt.

• Document length is found out using the unigram tokens generated

• C is computed using doc-termCount table , which contains each document\_ID and its document length (DL). Cqi for each term is calculated by computing each term and its frequency in the whole collection.

• The score for each document is calculated using the above formula and stored as a dictionary.

• The dictionary is sorted in the descending order based on the score generated

• The result is printed into a text file (Query-Likelihood-Ranking.txt) in the below format:

“query\_id Q0 doc\_id rank score system\_name”

➢ **Lucene Model: (using version 4.7.2):**

We downloaded and setup lucene library from <https://lucene.apache.org> . Lucene is widely used in both academic and commercial search engine applications.

We have added below three version.jar (libraries) in our java project.

* Lucene-core-VERSION.jar
* Lucene-queryparser-VERSION.jar
* Lucene-analyzers-common-VERSION.jar

➢ Implementation:

• Using “simpleAnalyzer” as our analyzer , we indexed the raw documents.

• Performed search for all the queries given in the project.

• The top 100 results for each query is printed into a text file (Lucene\_Ranking.txt)

in the below format: “query\_id Q0 doc\_id rank score system\_name”

**3.2 Task2- Phase1:**

➢ **Pseudo Relevance Feedback Model:**

We have implemented the Pseudo Relevance Feedback model for Query Enrichment using BM25 ranked results.

**➢ Implementation:**

• Select top (k=3) three ranked documents from BM25\_ranking result for each query

• We have removed the stopwords using common\_words.txt in those documents to avoid non-functional high frequency terms and then took top five high frequency terms form each document based on the justification we discussed before for parameter setting.

• For all those terms not present in individual query, we have expanded the query (retaining the order of query terms) using those five high frequency terms from each document.

• Result of new enriched query is stored in enrichedQueries.txt, which we will use in BM25 run.

• Rank documents again from BM25 score function using enriched queries.

**3.3 Query-by-Query Analysis:**

**➢** Analysis for Task 3 using three queries to compare the effect of stemming on ranking:

a) Query: code optimization for space efficiency

Query with stemming: code optim for space effici

Consider below example:

(DocID : Rank)

CACM-1947 :1 (unstemmed version) vs CACM-1947 :3 (stemmed version)

CACM-2748 :3 (unstemmed version) vs CACM-2748 :1 (stemmed version)

CACM – 1947: Initially CACM-1947 would only compare with files that contain the term “optimization” now after stemming it has to compare with all documents which contain word optimize, optimizations, optimal which are now reduced to stem class ‘optim’ and hence the rank falls for document CACM - 1947

CACM-2748: Before stemming CACM-2748 had no occurrence of ‘efficiency’ but after stemming the word ‘efficient’ has been stemmed to word effici creating more hits with query term hence its ranking rises.

b) Query: Parallel processors in information retrieval

Query after stemming: parallel processor in inform retriev

Consider below example:

CACM-2714 :1 (unstemmed version) vs CACM-2714 :3 (stemmed version)

CACM-1811 :6 (unstemmed version) vs CACM-1811 :1(stemmed version)

In the query and in the index all the occurrences of ‘processors’ have been stemmed to ‘processor’ so BM25 with stemming model picks up the documents that have the term ‘processor’, whereas BM25 model without stemming picks up only those documents containing the word ‘processors’. Document 2714 contains 3 mentions of the word ‘processors’ and Document CACM-1811 has 2 mentions ‘parallel-processor’ and ‘parallel-processors’ after stemming, CACM-1811 will have more hits than CACM-2714. So, its ranking goes up after stemming the corpus.

c) Query: Parallel algorithms

Query with stemming: parallel algorithm

Both the query seems to retrieve the same set of documents in terms of the ranking of documents along with some variations in ranking of specific documents. That is because both the stemmed and the unstemmed version of the query is the same, and the only variation is in the query term “algorithms” which is stemmed to “algorithm”.

Consider below example for the variation:

CACM-2714 :9 (unstemmed version) vs CACM-2714 :1 (stemmed version)

CACM-2714 had many occurrences of query term ‘parallel’, and has 5 mentions of word ‘algorithm’.

The un-stemmed BM25 would not match ‘algorithm’ with ‘algorithms’ hence the rank of CACM-2714 drops in the unstemmed version.

**3.4 Analysis for Extra credit task:**