Download and uncompress the following data from Canvas:

- `index_lucene_robust04_nostem.tar.gz`
- `index_lucene_robust04_krovetz.tar.gz`
- `index_lucene_robust04_porter2.tar.gz`
- `queries_robust04`: 249 search queries
- `qrels_robust04`: relevance judgments for the 249 queries

The `*.tar.gz` files are three indexes built for the same corpus (Robust04) using different stemming settings: the `n stem` index did not apply stemming; the `krovetz` index applied Krovetz stemming; the `porter2` index used Porter2 stemming (English Snowball). The other text processing settings are the same for the three indexes: texts were tokenized using Lucene’s `StandardTokenizer`; uppercase letters were transformed into lowercase ones; we did not remove any stop words when we built the indexes. Similar to HW1, we stored the docno of each document in the `docno` field, and we built indexes for the content of the document in the `content` field.

The Robust04 dataset was used in the TREC 2004 Robust Track\(^1\). The test collection includes 528,155 news articles and the human relevance judgments for 249 queries. We will perform experiments on this dataset in HW2.

Each line of `queries_robust04` is a search query, along with its query id. For example, the first line of `queries_robust04` is query#301 “International Organized Crime”. You can read the query file using `EvalUtils.loadQueries()`.

Each line of `qrels_robust04` stores the relevance judgment of a document with regard to a search query in the following format:

```
queryid 0 docno relevance
```

where `relevance` can take values 2 (`highly relevant`), 1 (`relevant`), or 0 (`not relevant`). For example, the first line of `qrels` is:

```
301 0 FBIS3-10082 1
```

\(^{1}\)http://trec.nist.gov/pubs/trec13/papers/ROBUST.OVERVIEW.pdf
which means the document FBIS3-10082 is relevant to the query #301. Note that in HW2, we consider both 2 (highly relevant) and 1 (relevant) as relevant documents in evaluation. You can read the qrels using EvalUtils.loadQrels() and EvalUtils.loadNrels().

Check out the HW2 starter code from GitHub:

- https://github.com/jiepujiang/CS5604_HW2

1 Comparing TF×IDF Variants (60 points)

As we discussed in Lecture 5, many best-match retrieval models rank results by a scoring function in the following form

\[
\text{score}(q, d) = \sum_{t \in q} \text{score}(t, d) = \sum_{t \in q} (\text{score}_{di}(t) \cdot \text{score}_{dd}(t, d))
\]

where \(\text{score}(q, d)\) sums up \(\text{score}(t, d)\) over each query term \(t\). In many retrieval models, \(\text{score}(t, d)\) includes two parts:

- \(\text{score}_{di}(t)\) is a document-independent weight for the query term \(t\) (such as the many variants of IDF).
- \(\text{score}_{dd}(t, d)\) is a document-dependent score with regard to both the query term \(t\) and the document \(d\) (such as the many variants of TF).

BestMatchSearch.DocAtATime offers a generic implementation of the document-at-a-time search algorithm. DocAtATime.search() performs a best-match search by taking the following inputs:

- a Lucene index reader object
- a document length reader object (see details in utils.DocLengthReader)
- a list of tokenized query terms
- a document-independent scoring function, as defined in DocumentIndependentWeight
- a document-dependent scoring function, as defined in DocumentDependentWeight

DocAtATime.search() is implemented in a generic way to accommodate different \(\text{score}_{dd}\) and \(\text{score}_{di}\) functions, which makes it flexible to finish some experiments in Problem 1 and 2. To help you get started, we have implemented a few example \(\text{score}_{di}\) and \(\text{score}_{dd}\) functions:
• **DocumentDependentWeight.BinTF** – the binary TF function:

\[
\text{BinTF}_{dd}(t, d) = \begin{cases} 
1 & \text{freq}(t, d) > 0 \\
0 & \text{freq}(t, d) = 0 
\end{cases}
\]

\(\text{freq}(t, d)\) is the frequency of the query term \(t\) in the document \(d\).

• **DocumentDependentWeight.RawTF** – the raw TF function:

\[
\text{RawTF}_{dd}(t, d) = \text{freq}(t, d)
\]

• **DocumentIndependentWeight.Uniform** – each term has an equal weight:

\[
\text{Uniform}_{di}(t) = 1
\]

• **DocumentIndependentWeight.IDF** – the original IDF with 0.5 smoothing:

\[
\text{IDF}_{di}(t) = \log \frac{N + 0.5}{n + 0.5}
\]

\(N\) is the total number of documents in the corpus, and \(n\) is the number of documents containing the query term \(t\).

### 1.1 Implementation (15 points)

Follow the example \(\text{score}_{di}\) and \(\text{score}_{dd}\) functions and implement the following new functions:

• **DocumentDependentWeight.LogTF** – the log TF function using a log base \(b\), where \(b\) is a parameter.

\[
\text{LogTF}_{dd}(t, d, b) = \begin{cases} 
1 + \log_b \text{freq}(t, d) & \text{freq}(t, d) > 0 \\
0 & \text{freq}(t, d) = 0 
\end{cases}
\]

• **DocumentDependentWeight.BM25TF** – the BM25 TF function, where \(k_1\) and \(b\) are two parameters. \(|d|\) is the length of the document \(d\) (the number of total words in the document); \(avdl\) is the average length of documents in the dataset. Note that in HW2, you do not need to count \(|d|\) and \(avdl\) by yourself. You can get \(|d|\) and \(avdl\) using \text{DocLengthReader.getLength()}\) and \text{DocLengthReader.averageLength()}\).

\[
\text{BM25}_{dd}(t, d, k_1) = \frac{(k_1 + 1) \cdot \text{freq}(t, d)}{\text{freq}(t, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avdl})}
\]

• **DocumentIndependentWeight.RSJ** – the binary independence model (BIM)’s term weight, also referred to as the RSJ weight (which is also the BM25 model’s IDF weight).

\[
\text{RSJ}_{di}(t) = \log \frac{N - n + 0.5}{n + 0.5}
\]
Table 1: Mean P@10 and AP (Average Precision) of 249 queries on the Robust04 dataset.

<table>
<thead>
<tr>
<th>P@10</th>
<th>Uniform</th>
<th>IDF</th>
<th>RSJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinTF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RawTF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = 2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = e$</td>
<td></td>
<td>0.337</td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = 10$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25, $k_1 = 1.2, b = 0.75$</td>
<td></td>
<td>0.426</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP</th>
<th>Uniform</th>
<th>IDF</th>
<th>RSJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinTF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RawTF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = 2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = e$</td>
<td></td>
<td>0.179</td>
<td></td>
</tr>
<tr>
<td>LogTF, $b = 10$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25, $k_1 = 1.2, b = 0.75$</td>
<td></td>
<td>0.238</td>
<td></td>
</tr>
</tbody>
</table>

Note that in Problem 1, you should not modify the definition of the two interfaces (DocumentIndependentWeight and DocumentIndependentWeight) or merge the posting lists by yourself. You only need to implement the scoring functions defined in DocumentIndependentWeight and DocumentIndependentWeight. DocAtATime will retrieve and rank results based on the score_dd and score_di functions. Include your implementations in your HW2 submission.

1.2 Experiments and Discussion

We hope to compare and understand the search effectiveness of different score_dd and score_di functions. Q1Example has some example program for evaluating the effectiveness of the four example score_di and score_dd functions on the Robust04 dataset. Q1Example retrieves the top 1,000 results for each of the 249 queries on the krovetz index using each combination of the example score_di and score_dd functions. Q1Example evaluates each query’s top 1,000 search results by P@10 and average precision (AP). Then, it reports the mean (average) values of P@10 and AP over the 249 queries for each combination of the score_dd and score_di functions.

1.2.1 Compare Different Scoring Functions (15 points)

Read Q1Example. Write your own experiment program to evaluate each combination of the following score_dd and score_di functions on the krovetz index:

- 6 score_dd functions: BinTF, RawTF, LogTF ($b = 2$), LogTF (natural log base), LogTF ($b = 10$), and BM25 ($k_1 = 1.2, b = 0.75$).
- 3 score_di functions: Uniform, IDF, and RSJ.
Similar to Q1Example, your experiment program needs to report the mean P@10 and AP of the 249 queries (only considering the top 1,000 search results) for each combination of the scoring functions. There are 6 × 3 = 18 combinations in total. You need to include and discuss the experiment results:

- Report the mean P@10 and AP in tables similar to Table 1. We put some numbers in Table 1 to help you debug your implementations.
- Discuss the differences of the scoring functions in terms of their search effectiveness (by mean P@10 and AP over the 249 short queries) based on your experiment results.

Include your experiment program in your HW2 submission. Note that in Problem 1.2 you should not need to read the query or qrels files by your own or write your own P@10 and AP evaluation functions (please simply use the existing ones in the starter code).

1.2.2 Parameters (15 points)

As you may have noticed in 1.2.1, the parameters of LogTF<sub>dd</sub> and BM25<sub>dd</sub> have some influence on the effectiveness of the search results. Write a program to output the mean P@10 and AP of 249 short queries (only considering top 1,000 search results) on the Krovetz stemming index for the following parameters:

- LogTF<sub>dd</sub>: \( b \) from 2 to 20 with step 2. Set the Score<sub>di</sub> function to IDF.
- BM25<sub>dd</sub>: \( k_1 = 0.5, 0.75, 1.0, 1.2, 1.5, 2.0, 2.5, \text{ or } 3.0 \). \( b = 0.5, 0.75, \text{ or } 1.0 \). Set the Score<sub>di</sub> function to RSJ.

In your report, you should plot the results to make it easier to read (e.g., using x-axis for different parameter values and y-axis for the mean P@10 or AP over the 249 short queries). In addition, discuss the influence of the parameters on search effectiveness (mean P@10 and AP) and the appropriate or optimal range of the parameters based on your experiment results. Include your experiment program in your submission. You do not need to submit your program (if any) for plotting the results.

1.2.3 Stemming and Search Effectiveness (15 points)

So far we have only performed experiments using the Krovetz stemming index in Problem 1.2.1 and 1.2.2. Rerun your experiments in 1.2.1 and 1.2.2 using the nostem and the porter2 indexes. Report the results (mean P@10 and AP) on the nostem and the porter2 indexes and compare with those using the krovetz index. Discuss the following issues:

- Which of the three stemming options yield the best search effectiveness on the Robust04 dataset (in terms of P@10 and AP)?
- Does the particular choice of stemming option influence your conclusions in Problem 1.2.1 and 1.2.2 (such as which score<sub>di</sub> and score<sub>dd</sub> function is the best and the appropriate range of the parameters)?
2 Document Length and Relevance (20 points)

We have examined many variants of TF functions in Problem 1. Some of them take into account document length (such as the BM25TF function), while others do not. We examine the relationship between document length and relevance in Problem 2.

2.1 Relevance Judgments (10 points)

For each of the 249 queries, read the sets of judged relevant results \((\text{relevance} = 1 \text{ or } 2)\) and non-relevant results \((\text{relevance} = 0)\) from \texttt{qrels\_robust04} using \texttt{EvalUtils.loadQrels()} and \texttt{EvalUtils.loadNrels()}. Compute and report the average lengths of the judged relevant and non-relevant results over the 249 queries. For example, you can plot the results like Figure 1. Note that you can use \texttt{DocLengthReader.getLength()} to retrieve the length of each document.

Discuss the following question based on your results and plots:

According to the human relevance judgments, are shorter documents more likely relevant in the Robust04 dataset?

2.2 Retrieval Models (10 points)

We have retrieved the top 1,000 results for 18 different combinations of TF×IDF functions in Problem 1. Now we can examine if these TF×IDF functions tend
to retrieve longer or shorter documents by looking into the length of results at different ranks. For example, we can compare the average length of search results in top 100 or 200 with those at lower ranks such as from rank 500 to 1000. We can also examine if there is a significant correlation between the rank of a retrieved result and its length.

Examine and discuss the following questions:

• Does the Binary TF function tend to retrieve longer documents? Justify your answer using data and try to explain why.

• For the LogTF function, does a smaller log base \( b \) favor longer documents by a greater extent comparing to a greater \( b \)? Justify your answer using data and try to explain why.

3 Term-at-a-time Search (20 points)

The WBC textbook (on pages 166–169) introduced two basic strategies of processing (e.g., intersecting or unioning) multiple posting lists. We also discussed the two strategies in Lecture 4.

• The **document-at-a-time** strategy (page 167) keeps a cursor for each posting list. It compares the current cursors, finds the cursor(s) pointing to the document with the smallest ID, computes the score (such as TF × IDF) of that document, and moves these cursor(s) to the next position(s). The starter code provides an implementation in BestMatchSearch.DocAtATime.

• The **term-at-a-time** strategy (page 169) processes the posting lists one after another. It stores the temporary scores of results in memory. Each time it sequentially reads a term’s posting list from the beginning to the end and updates the temporary scores while iterating over the posting list.

The following is an example to help you recap the term-at-a-time approach. Let us assume a query: **retrieval query reformulation**. The posting lists of the three query terms are as follows:

• \( t_1 = \text{retrieval} \)

<table>
<thead>
<tr>
<th>docid</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

• \( t_2 = \text{query} \)

<table>
<thead>
<tr>
<th>docid</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

• \( t_3 = \text{reformulation} \)

<table>
<thead>
<tr>
<th>docid</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Let us just assume that we are using a simple TF×IDF scoring function to rank search results, where TF is the raw term frequency, and the terms’ IDF values are:

- IDF(retrieval) = 3.0
- IDF(query) = 2.0
- IDF(reformulation) = 4.5

If we process the posting lists by the sequence \( t_1, t_2, \) and \( t_3 \), the term-at-a-time algorithm runs as follows:

1. **Loop 1: \( t_1 = \text{retrieval} \)**
   After processing \( t_1 \), we should have the following temporary scores:

<table>
<thead>
<tr>
<th>docid</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp score</td>
<td>6.0</td>
<td>3.0</td>
<td>12.0</td>
<td>9.0</td>
<td>6.0</td>
</tr>
</tbody>
</table>

2. **Loop 2: \( t_2 = \text{query} \)**
   After processing \( t_2 \), we should have the following temporary scores:

<table>
<thead>
<tr>
<th>docid</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp score</td>
<td>2.0</td>
<td>6.0</td>
<td>5.0</td>
<td>4.0</td>
<td>14.0</td>
<td>4.0</td>
<td>11.0</td>
<td>8.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

3. **Loop 3: \( t_3 = \text{reformulation} \)**
   After processing \( t_3 \), we should have the following final search result scores:

<table>
<thead>
<tr>
<th>docid</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp score</td>
<td>2.0</td>
<td>6.0</td>
<td>27.5</td>
<td>4.0</td>
<td>14.0</td>
<td>13.0</td>
<td>11.0</td>
<td>21.5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

- The temporary scores are often stored as a hash table or just as a list sorted by document IDs. To get the top \( k \) highest-scored results efficiently, we can use a priority queue to avoid sorting all the scores. You can find some examples of using PriorityQueue in `BestMatchSearch.DocAtATime`.

### 3.1 Implementation (10 points)

The WBC textbook (on page 169) suggests using a hash table accumulator to store the temporary scores. Please implement the term-at-a-time approach in `BestMatchSearch.TermAtATime` based on this implementation suggestion. You can use either HashMap or HashTable in Java. The two classes are very similar, except that the former one is not thread safe. Make sure your implementation follows the `BestMatchSearch` interface. Include your implementation of `TermAtATime` in your HW2 submission.

You can verify your `TermAtATime` by using it to run the experiments in Problem 1. It should produce very similar results to those by using `DocAtATime`. Note that due to the existence of search results with the same scores (ties), your `TermAtATime` and `DocAtATime` may have slightly different rankings of search results. But the mean P@10 and AP over the 249 queries should not be much different).
3.2 Experiment and Discussion (10 points)

Q3Example measures the efficiency of the DocAtATime method (how much time it takes to retrieve results). The experiment retrieves the top 1,000 search results for each of the 249 queries on the Krovetz stemming index (the search results are ranked by RawTF×IDF). It records and reports the total time for processing the 249 queries. It runs the experiment 10 times, reporting the total time for each round and the mean and standard deviation (std.dev) of the 10 rounds.

Please follow Q3Example and write your own experiment program to compare the efficiency of DocAtATime and TermAtATime. Report your results and discuss which one is faster on the Robust04 dataset.

A A Checklist for Your Submission

- A report including answers to all questions (in pdf).
- Source code (include a brief readme with your submission if necessary).

Pack all the stuff as a .zip or .tar.gz file and upload to Canvas’s HW2 submission link. If you decide to use the 5-day extension (you can only use it once during the whole semester), send an email and let the instructor and the TA know.