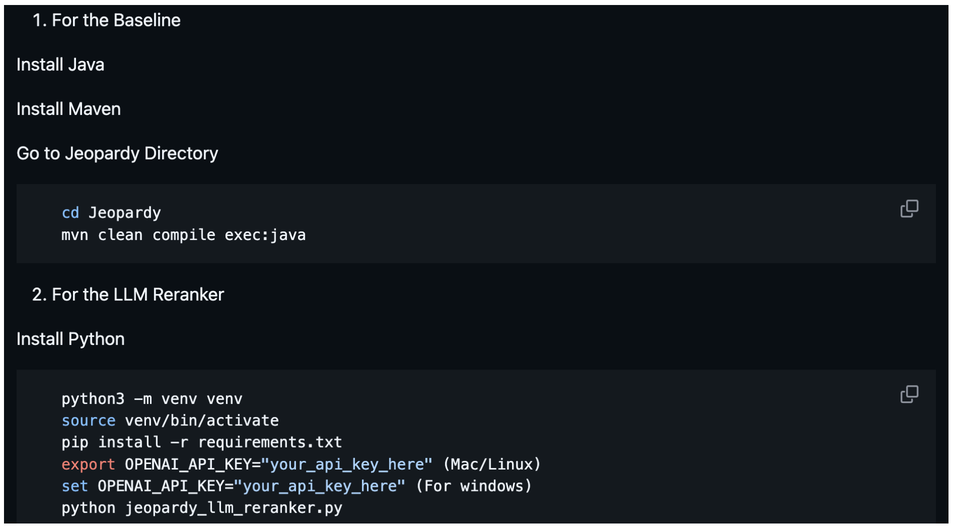
Jeopardy IR Project

Members: Amimul Ehsan Zoha, Taoseef Aziz
GitHub Repo Link: https://github.com/amimul1/IR_Amimul_Taoseef.git

How to run our project: (Available in ReadMe.md of our github repo)



1. For the Baseline

Install Java

Install Maven

Go to Jeopardy Directory

```
cd Jeopardy
mvn clean compile exec:java
```

2. For the LLM Reranker

Install Python

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
export OPENAI_API_KEY="your_api_key_here" (Mac/Linux)
set OPENAI_API_KEY="your_api_key_here" (For windows)
python jeopardy_llm_reranker.py
```

You can tweak the text-processing settings by editing the MyAnalyzer class -just comment out any filters you'd like to disable. For instance, if you want to see results without removing stopwords, comment out the .addTokenFilter(StopFilterFactory.class, stopMap) line (around line 194). The results of different configurations are discussed in the next section.

Performance

We used MRR and P@1 metrics to report the scoring of our system. Justifications behind the choice:

MRR: We used it extensively during the development process to keep track of our progress and evaluate our system. When we only have one good answer for a given query, a good measure to use is Mean Reciprocal Rank. It is valuable for evaluating the performance of a Jeopardy QA system as it averages the reciprocal ranks of the correct answers across multiple queries, providing a broader perspective on overall system performance.

P@1: Precision at 1 is basically a fancy way of saying how many of the top results (first predicted answer) out of the total questions are correct. It is a very good performance measurement metric in the Jeopardy game since the accuracy of the top result is critical since there's only one chance to respond. (so we have to know how many answers of the total questions we got correct at the top position).

Configuration	Stop Words Included, With Porter Stemmer, Duplicates Included	Stop Words Included, With Porter Stemmer, Duplicates Removed	Stop Words Included, Without Porter Stemmer, Duplicates Included	Stop Words Included, Without Porter Stemmer, Duplicates Removed	Stop Words Removed, With Porter Stemmer, Duplicates Included	Stop Words Removed, With Porter Stemmer, Duplicates Removed	Stop Words Removed, Without Porter Stemmer, Duplicates Included	Stop Words Removed, Without Porter Stemmer, Duplicates Removed
MRR	0.36	0.33	0.37	0.25	0.36	0.32	0.36	0.24
P@1	0.29	0.25	0.29	0.19	0.3	0.26	0.28	0.18

The choice of text processing techniques in building the QA system has a significant impact on its performance
Best Configuration for P@1: For achieving the highest precision at the first rank, the optimal configuration seems to be using Porter Stemmer, removing stop words, and not removing duplicates. This configuration maximizes the chances that the top result is correct, crucial for applications like a Jeopardy game where the first answer's accuracy is paramount. In this configuration, the number of answers that were somewhere in predictions = 49 / 100
Best P@1 Value: 0.29
Best MRR Value: 0.36
Best Configuration for MMR: Including stop words, using Porter Stemmer, and not removing duplicates.

We can see that using Porter Stemmer increased our scores, stemming might help in retrieving relevant documents by reducing words to their base forms encouraging more matches where there maye have been misses.
Removing duplicate words from the indexing process consistently lowers both MRR and P@1 across all configurations. This indicates that duplicates might be providing important reinforcement that help the retrieval algorithm prioritize relevant documents, possibly by reinforcing key topics or terms within the documents.

Error Analysis

Keyword Match:
Many questions that are correctly answered have direct matches in the text. If a query term exactly matches some key terms in the text documents being searched, the system can retrieve the correct answer.

No understanding of pragmatics: Some questions that the system fails to understand the context of or intent behind a question, leads to wrong answers.

The query-answer pair in the gold is something that would typically be supplied for instruction tuning for transformers: “We'll give you the museum. You give us the state.” Our system is not intelligent enough to understand that this is wanting the name of the state.
The correct answer was "Idaho," but the system predicted "Red Jacket" who was a seneca orator and the wikipedia article has references to museums and art but this is clearly the wrong article. There is no pragmatic sense in the system, just keyword identification.

Example of where it did not retrieve the desired article in the top 10 at all:
[content]
STATE OF THE ART MUSEUM (Alex: We'll give you the museum. You give us the state.)
The Sun Valley Center for the Arts[content]
Idaho[content]
[Red Jacket, Terre Haute, Indiana, Narasimha, Bhavacakra, Edward Hopper, Chinatown, Los Angeles, Louisiana State University, Phoenix, Arizona, My Neighbor Totoro, Northern Arizona University]

Lack of information in the original wikipedia entry:
The wikipedia article for the gold, which is "Idaho" has zero mentions of "art" and two mentions of "museum" in the references, so this document is expected to have a poor score, so gets out competed by other documents. There is no information in this article at all about whether Idaho hosts this particular museum to the human reader either, so it is hard to expect a retrieval model to do well here either. The token "state" is probably pretty common in the document collection so has very little impact in the retrieval even though "state" appears 9 times on the wikipedia article for Idaho. Red Jacket's wiki article has 2 mentions of "art".

Example of Poor ranking, but retrieved (in top 10): The article for Crest (toothpaste) has 43 mentions of the token "Crest" and 5 mentions of the phrase "Crest Toothpaste". In the wikipedia article for P&G there is only 2 mentions of the token "Crest". In this case, BM25 scores the article for Crest higher than the article for P&G which is the gold answer. There is **no contextual/relational understanding** of the applicable class of named entity that is being asked for (Company/Business) and no relational understanding between the current company and the parent within the retrieval system at all.

Example:
[content]
NAME THE PARENT COMPANY Crest toothpaste[content]
Procter & Gamble[content]
[Crest (toothpaste), Toothpaste, Procter & Gamble, Air India (football club), Trust in Luton, Great North Eastern Railway, 186th Infantry Regiment (United States), Jane Wyman, The Age, Kagu]

Example of when it did well:

[content]
AFRICAN CITIES Wooden 2-story verandas in this Liberian capital are an architectural link to the U.S. south[content]
Monrovia[content]
[Monrovia, Jug Tavern, Liberia, History of Liberia, Glenview Mansion, Americo-Liberian, Architecture of India, William R. Tolbert, Jr., Ellwood House, Delavan Terrace Historic District]

In the wiki article for Monrovia, The token African appears 8 times, "Cities" appears 2 times, Liberian shows up 9 times (the stem Liberia appears 35 times), capital shows up 4 times leading up to a high BM25 score (high f(q,D) == document frequency). The token "architectural" appears once in each of the top 2 results; (the stem "architectur" appears twice in the second hit). This contributes to a large IDF in the BM25 calculation, which is similar to the TF-IDF calcualtion.

Re-ranking top results using LLM for performance boost

Our Lucene search often fails to retrieve the correct page in its top-10, retrieving the correct article in only ~50 % of queries at k = 10, capping Precision@1 at 0.49 no matter how sophisticated the reranker.

So, by expanding k, we (i) widen the candidate pool so the answer is more likely to appear somewhere in the list (increase p@k), then (ii) we let GPT-4o-mini semantically rerank those candidates.

k (candidates given to GPT-4o-mini)	Found %	Baseline P @ 1	LLM P @ 1	Baseline MRR	LLM MRR
10	49	0.30	0.48	0.37	0.48
20	54	0.30	0.53	0.37	0.53
30	56	0.30	0.54	0.37	0.55
40	59	0.30	0.56	0.37	0.57
75	65	0.30	0.61	0.37	0.62

Fig: Performance with LLM Reranking for different values of k (no. of candidates)

Even with k = 10, GPT raises P @ 1 from 0.30 -> 0.48 purely by better ordering the same candidates. This proves that LLMs are well suited for this task. With the answer already present (k = 10), GPT elevates it to rank 1 in most cases, that's why MRR and P@1 are the same.

Recall drives gains: as k grows, answer found in the candidates list increases and final accuracy climbs. The jump of k from 10->20 recovers 5 extra answers and delivers a 5% leap in P @ 1 resulting in a respectable value of p@1 of 0.53. Similar but smaller returns appear up to k ≈ 40. We tried setting k to a very big value to see if the performance further improves, and we ended up getting p@1 = 0.61 and MRR =0.62 for k =75.

However, there is a trade-off. Every 10 extra titles cost more prompt tokens, making our system slower and more costly. So, depending on the use case, we can choose a sweet spot value of k.

Example output:

```
python jeopardy_llm_reranker.py
Reading queries file...
Loaded 100 questions

Baseline Metrics:
MRR = 0.3685
P@1 = 0.3000
Found = 54 / 100

Reranking predictions using GPT-4o-mini...

Reranked Metrics:
MRR = 0.5320
P@1 = 0.5300
Found = 54 / 100

Reranked results written to src/main/resources/queriesReranked.txt
```

Implementation overview

We built our Jeopardy! QA engine on top of Lucene. It reads the Wikipedia article files and the question list, then creates an in-memory inverted index. Queries are scored and ranked using Lucene's default BM25 (an enhanced TF-IDF) algorithm to surface the most relevant article titles. We also incorporated Wikipedia's category labels by appending them to each article's text This way they contribute to relevance signals without overpowering the core content. We did the same thing on the query side, by prefixing each question with its category.

Important Note on class MyAnalyzer

Constructs and returns the custom analyzer which uses the mentioned preprocessing steps in the pipeline for both the queries and documents. We also observed that the wiki articles contain artifacts like hyphenated words, subheadings marked with double =, double hyphens, and <tpl> tags. To handle these consistently, we strip or normalize each of those patterns in both the indexed text and the incoming queries. That way, any filtering we apply on the Wikipedia side is mirrored exactly when processing the user's questions, and vice versa.
We normalized both the documents and the queries through a combination of explicit replacements and a custom analyzer chain.

replaceAll() preprocessing: Strip out problematic sequences likedouble hyphens (--), exclamation marks (!), double-equals (==), and <tpl> tags-by replacing them with either a space or empty string. This prevents Lucene's query parser from mis-tokenizing or throwing EOF errors. Custom Analyzer filters (applied in order):
LowerCaseFilterFactory: Lowercases all tokens for case-insensitive matching.
StopFilterFactory: Removes common stopwords using our stopwords.txt list.
HyphenatedWordsFilterFactory: Splits hyphenated words into separate tokens (e.g., "state-of-the-art" → "state", "of", "the", "art").
KeywordRepeatFilterFactory: Duplicates each token to boost its weight during scoring, ensuring important terms carry more influence.
SnowballPorterFilterFactory: Applies Porter stemming so that different morphological forms (e.g., "running", "runs") map back to a common root.

By mirroring these steps on both the Wikipedia text and the incoming queries (ensuring that we do similar things on both indexing side and query side), we maintain consistent tokenization and matching across our entire retrieval pipeline. We experimented with additional techniques like removing duplicates but we left out the ones which hurt our scores.