CS 483/583: Programming Project Building (a part of) Watson Report

Aryam Gomez, Amimul Ehsan Zoha, Muaz Ali

GitHub Repo Link: https://github.com/LucodeProjects/TextRetrevalProjectPrototype

Commands to Run the Project

Watson Engine:
The simplest is to load the project in IntelliJ IDEA and run the WatsonEngine.java file.

Alternatively:
java -classpath {all the libraries needed} org.WatsonEngine, or you can also access the README.md file and use the Run option from there.

LLM Evaluation:
Open the JupyterNotebook named as LLMEvaluation.ipynb to access the LLM evaluation part.

The current configuration is set to the combination which gives the best results for our core implementation:
TPL Tags Removed, Stop Words Removed, Using Porter Stemmer, Duplicates Included

If you want to change the configurations, you can modify the Custom Analyzer: MyAnalyzer by commenting out the options. For example, if you want to see the scores without using Porter Stemmer, do: `//.addTokenFilter(StopFilterFactory.class, stopMap)` in line 293 of WatsonEngine class.

Code Description

Private static void loadData(String directory, IndexWriter writer)

- This function is incharge of parsing all the documents containing wiki pages, which are found in the ‘directory’. Then as it parses the file it separates the titles from the text and adds it to the index using the ‘writer’.

Private static void addDoc(IndexWriter index, String title, String text)

- This function does the actual writing of the wiki page and title to the index. Before that it will also clean the text a bit and concatenate the title to reflect the approach described below in the Core Implementation section.

Private static HashMap<String, String> getQueryQuestions(String file)

- Parses the questions file at the location ‘file’ and creates a HashMap mapping question/clues to the answers. Here the category of the question is concatenated at the beginning of the query following our approach described below in the Core Implementation section.

Private static List<String> queryIt(String query)

- Takes the query passed in as ‘query’ and applies some cleaning to it, described in the Core Implementation section below. It then uses the Lucene system to perform the search to then return the top 10 results as a list of strings.

Private void queryAndComputeStats(HashMap<String, String> queryAnswers)

- This function iterates through the collection of queries and answers, executing the queries, and performing calculations for both the MRR and P@1 metrics as it goes.

Class MyAnalyzer - public Analyzer get()

- Constructs and returns the custom analyzer which uses the StandardTokenizerFactory, and as filters it uses the LowerCaseFilterFactory, HyphenatedWordsFilterFactory, KeywordRepeatFilterFactory, and the SnowballPorterFilterFactory.

Core Implementation

Choosing Lucene we parsed through the files and the file containing the questions to make and an index to query against. In terms of using the categories we decided to not use them outside of just including them in the text of the wiki page itself. This way the categories are still involved but not on their own. Similarly, in the query side we concatenated the category to the beginning of the question. This was done before any of the processes described below took place.

We did notice some features about the wiki pages that we decided to find a potential solution for. Features about the text that stood out to us were hyphenated words, subheaders with double ‘=’ symbols, double hyphens, and ‘tpl’ tags. While the queries did not have double ‘=’ symbols or ‘tpl’ tags, hyphenated words, and double quotes were also found in the query

text. So if we applied a filter to the wiki page text we also equally applied to the query, and vice versa.

We handled these things in 1 of two ways; either by using a replaceAll() call or using a corresponding filter within our custom analyzer for the Lucene index. Features like double hyphens, exclamation marks (these somehow cause an EOF exception to be thrown out from the Lucene parser at query time from the queries specifically), double '=' symbols, and 'tpl' tags are handled in the code using a replaceAll() call. Replacing it with either a space or an empty string with the intention to separate words so that they would not get processed as one token by the analyzer. The second way we handled some of the cases was through our custom analyzer. In this analyzer we decided to use 4 additional filters in a very specific order that is as follows;

- LowerCaseFilterFactory - Filters the text so that it is all cased lower.
- HyphenatedWordsFilterFactory - Splits hyphenated words into multiple tokens (default config)
- KeywordRepeatFilterFactory - Introduced a weight based on the importance of the token, to give it more impact during search.
- SnowballPorterFilterFactory - Uses the Porter Stemming Algorithm to further process tokens.

Finally, we also considered tactics like removing stopwords, removing duplicate tokens, using filters for accented words, using filters for english words in the possessive form, and even removing the 'tpl' tags along with the text between the opening and close tags. However these did not make the final cut because they either had no effect or a detrimental effect to our performance metrics.

Measuring Performance

We used MRR (For dev process and system evaluation) and P@1(For final report and showing to customers) 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).

Combinations				MRR	P@1
TPL Tags Removed	Stop Words Included	With Porter Stemmer	Duplicates Included	0.3602	0.29
			Duplicated Removed	0.327	0.25
		Without Porter Stemmer	Duplicates Included	0.3656	0.29
			Duplicated Removed	.248	0.19
	Stop Words Removed	With Porter Stemmer	Duplicates Included	0.3637	0.30
			Duplicated Removed	0.3226	0.26
		Without Porter Stemmer	Duplicates Included	0.3576	0.28
			Duplicated Removed	0.236	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: 0.30

Best Configuration for MMR: Including stop words, using Porter Stemmer, and not removing duplicates.

Best MRR Value: 0.3637

We can see that using Porter Stemmer increased our scores, stemming might help in retrieving relevant documents by reducing words to their base forms.

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 contextual signals that help the retrieval algorithm prioritize relevant documents, possibly by reinforcing key topics or terms within the documents.

Error Analysis

Some patterns we observed:

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.

Lack of Contextual Understanding:

Misinterpretation of Key Details: Some questions that fail to understand the context or intent behind a question, leads to wrong answers.
Misinterpretation:
Example: "STATE OF THE ART MUSEUM (Alex: We'll give you the museum. You give us the state.) The Taft Museum of Art"

In the query there is a key hint: “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.

Analysis: The correct answer was "Ohio," but the system predicted "Museum of Science and Industry (Chicago)," which indicates a misunderstanding of a key part of the query.
.

Lack of information in the original wikipedia entry:

For example:
The hint: “TIN" MEN This Italian painter depicted the "Adoration of the Golden Calf"

Answer: Tintoretto

The wikipedia entry for Tintoretto does not contain any words (information) that suggest that he depicted the Adoration of the Golden Calf painting. The entry does contain the quote "Adoration of the Golden Calf".

Hence, our system also suffers due to lack of information.

Leveraging LLMs to improve performance (Graduate)

To leverage Large Language Models (LLMs) in our system, we re-ranked the responses generated by our Engine using LLMs. We picked two mainstream LLMs for our analysis, Gemini and ChatGPT (3.5). Based on this re-ranking, we evaluated the performance.

	Baseline	ChatGPT (GPT 3.5)	Gemini
Precision@1	0.29	0.65	0.64
MRR	0.36	0.67	0.66

This table shows the performance of the queries with LLM re-ranking.

Analysis:
We observe a noticeable improvement in the performance of the system using LLMs. We make the key observations:

- Precision@1 becomes (almost) equal to MRR, meaning that the correct answer was brought to the top for almost all the queries if it was present in the initial ranking of the system. This is because MRR and Precision@1 become equal if for all the queries that the correct answer is present in the responses, the answer is the top ranked answer.
- The gain in performance is almost double for the metrics we used. This is likely due to the fact that LLMs have been trained on the information related to the queries. The training data includes the information needed to correctly answer the queries.
- This high performance gain is indicative of the fact that LLMs are suitable for the tasks of solving trivia games like jeopardy.
- ChatGPT is (ever so) slightly better than Gemini.

Additional Information:

For 100 queries in the original questions file, we created batches of 10 (each containing 10 ranked responses generated by our Engine) to be sent to LLM, to optimize for response generation time and token limit.

Query format to LLM:

Can you re-rank the answers for the following queries generated by our engine?
Please only output a list of strings as text like: [str1, str2, str3], [str1, str2, str3]. Give code as a python list.

Query: abc…
Query: xyz…

Sample Output

******** Welcome to our Watson Engine! ********
... Loading files for indexing from src/main/resources/wiki-subset-20140602/
Total Wiki Pages Loaded: 280794
... Loading questions file: src/main/resources/questions.txt

MRR = 0.3637896825396825
P@1 = 0.3
Answer was somewhere in predictions = 49 / 100
Queries and their output has been written to: src/main/resources/queriesProcessed.txt