Caroline O'Neill
12/6/21
CSC 483

Watson Project:

https://github.com/c-oneill/csoneill-watson-project

Note: See README for usage and overview documentation. Results summarized here.

1. Indexing and Retrieval

The main logic for indexing occurs in the Indexer.java class. Each Indexer object takes parameters to turn on/off stemming, lemmatization, and stop words. Since Lucene doesn't provide built-in lemmatization, I utilized CoreNLP's library.

Each Wikipedia text file was parsed line by line, aggregated by article, lemmatized, and then built into a Lucene Document object. Each Document had three fields – content, title, and categories (pulled directly from each line "CATEGORIES: …").

Tokenization and porter stemming can be performed by one of Lucene's pre-packaged analyzers. But since I used CoreNLP for lemmatization, I created a custom analyzer so I could 1) turn on/off stop word filtering and 2) use the standard tokenizer while avoiding stemming already lemmatized words. In addition, I wrote a custom filter factory for removing tokens. I didn't end up using it but it might be useful for future features.

parsing Wikipedia content:
It wasn't an issue for most articles, but the same characters used to denote a title, "[[..]]", also denoted other in-text files, "[[File…". Before separately handling this case, some articles were split prematurely.

query building:
I used the full body of the clue to query the content field while simultaneously querying the category field. The Jeopardy clues contain many of the same metacharacters used for Lucene query parsing; I opted to simply remove Lucene metacharacters from the query body. In future improvements I'd like to add better support for direct quotation matching.

2. Performance

Since Jeopardy looks at just one answer per clue, I only considered one document relevant per query. Arguably P@1 is the only metric that matters for in-game performance. But I also looked at Mean Reciprocal Rank as a kind of "soft P@1" so I had a better gauge of improvement through development.

Below I've listed the index/search parameter combinations in order of decreasing performance on the provided set of 100 queries:

1) JMStemIndex:
stemming, no lemmatization, stopwords kept, Jelinek-Mercer language model (lambda = 0.1)
total correct: 33
MRR: 0.4121822231244155

2) StemIndex:
stemming, no lemmatization, stopwords kept, BM25 cosine similarity
total correct: 20
MRR: 0.29660469719972476

3) PlainIndex:
no stemming, no lemmatization, stopwords kept, BM25 cosine similarity
total correct: 21
MRR: 0.2847734569290168

4) StemNoStopIndex:
stemming, no lemmatization, stopwords removed, BM25 cosine similarity
total correct: 19
MRR: 0.26857287821013204

5) LemmaIndex:
no stemming, lemmatization, stopwords kept, BM25 cosine similarity
total correct: 18
MRR: 0.2637382902354505

6) PlainNoStopIndex:
no stemming, no lemmatization, stopwords removed, BM25 cosine similarity
total correct: 16
MRR: 0.2397028004576838

7) LemmaNoStopIndex:
no stemming, lemmatization, stopwords removed, BM25 cosine similarity
total correct: 14
MRR: 0.21972376175403216

8) TFIDFStemIndex:
stemming, no lemmatization, stopwords kept, TF-IDF similarity
total correct: 1
MRR: 0.02895343810404578

3. Scoring Function

On the best performing system so far – porter stemming, stop words kept, BM25 similarity – I swapped out the Lucene Similarity object under the hood. Instead of the BM25 probabilistic framework I tried the ClassicSimilarity TF-IDF vector space model and the Jelinek-Mercer Language Model.

TF-IDF ranking performed significantly worse, even without BM25 tuning – only correctly answering one clue.
On other hand, a bag-of-words language model with Jelinek-Mercer smoothing performed measurably better than BM25. Though this isn't necessarily a fair comparison; I optimized the Jelinek-Merced lambda parameter by checking increments of 0.1, whereas no tuning was performed for BM25. Interestingly the ideal lambda value was 0.1, which is ideal for long queries, creating a "disjunctive like" search.

4. Error Analysis

My best performing system used porter stemming, no lemmatization, stop words, a bag-of-words language model, and tuned Jelinek-Mercer smoothing. It answered 32 out of 100 clues correctly with a Mean Reciprocal Rank of about 0.4 (or on average the correct answer was ranked about 2.5).
Interestingly, the lemmatized search was not only outperformed by porter stemming, but also underperformed plain search (no stemming, no lemmatization). This could possibly be a result of certain search words that might differentiate a result being transformed into a more common base word. Subsequently the correct answer would drown in too many hits returned.

common error classes:
1) direct quotation matching. A few correct answers are ranked more than a couple hundred and even more than 10,000 hits down. One common thread is these clues provide a direct quotation that should identify the individual/media, but my system does not provide sufficient quote matching

2) "Alex: We'll give you the…". My system does not properly parse these portions. It follows that my system returns a related answer, but not in the correct category (i.e., returning a museum, not the museum the state is in).

3) dates. Several queries would match on key concepts but, for example, would identify the wrong actor working on a similar project in a different year. I can't isolate dates as the exact cause, but better support for dates might clear up some inconsistencies.

4) broad vs. niche topics. The system seems to perform better on topics that are narrower. I suspect broader articles are more likely to cover multiple aspects of the topic, throwing off keyword weights from the query. Slavery and Giant Panda, for example, aren't found within 10,000 hits.