CSC 583: Programming Project 1

Andrew Zupon

1 Code Description

- Make sure all wiki test docs are in `src/main/resources/wiki-subset-20140602` if you want to build an index.
- Unzip the indexes you want to use in `src/main/resources/`
- The indexes will only build if they don't already exist. To build the index, choose which kind you want ("stems", "lemmas", or "plain") on line 52 of `zupon_andrew_project.scala` and run `sbt run`. This will take between 1 and 2 hours depending on the index and hardware.
- To test the system using the "stems" index (the best performing one), run `sbt test`. This will print results for the naïve and improved models using default and modified similarity functions. This should take under a minute. If the index doesn't already exist, this will try to build it (and take a long time)!
- If you want to test the model with different indexes than "stems", you'll need to uncomment lines in `QueryEngineTest.scala` between approximately lines 169 and 212.

2 Indexing and Retrieval

This section describes how I built the various indexes used for retrieval, along with how I built the queries from the Jeopardy questions and categories.

First I will discuss how I preprocessed the Wikipedia documents. The first step was to gather the directory into a list of files, then loop through each file. Within each file, I joined the lines into one long string and then split on `\n\n[[`, since each Wikipedia page title is enclosed in double brackets and follows the preceding page after two empty lines. This turned the file string into an Array, where each entry in the Array was a Wikipedia page. Next, I looped over each entry in the array and performed various regex replacements to eliminate "junk" from the Wikipedia dump. This included hyperlinks as well as metadata about files, images, and references. From there, the processed text was funneled to the indexes.

I ended up building three different indexes: one that used stemming (`indexStems`), one that used lemmatization (`indexLemmas`), and one that used neither stemming nor lemmatization (`indexPlain`). For all three indexes, I split each line (which is an entire Wikipedia page and its title) on a dummy separator symbol that I inserted during preprocessing. The head of the split is the `pageTitle`, and will get written into the `'docid'` field with the Lucene writer. The tail of the split is the `pageText`. This text will ultimately get written into the `'text'` field, but how it is processed depends on the specific index.

For the stemming index, which uses the StandardAnalyzer, I simply write the entire `pageText` to the index; the StandardAnalyzer takes care of the rest. For the lemmatizing and plain indexex, which use the WhitespaceAnalyzer, I need to do additional processing on the text. To get lemmas, I use the CoreNLP Processor to tag and lemmatize each sentence of the text, extract the lemmas, and join them as a string with a single space character in between. For the plain index, which does neither stemming nor lemmatization, I also use the CoreNLPProcessor to extract the individual tokens of each sentence before similarly joining them with spaces. In hindsight, I probably could have done this part more efficiently, but the way my code was structured up to this point funneled the "plain" text in this way.

For this initial naïve approach, beyond the regex filtering of hyperlinks and other metadata, I did

not filter out any stop words from the text, nor did I do any additional processing on the `'docid'` field.

Next I will discuss how I build the query from the *Jeopardy!* question. Again, I started with some minor preprocessing. I split the file on empty lines, so each line contained the question category and the clue itself. I then simply concatenated all the words in the category and clue into a string and fed them to Lucene as an OR query. For the queries, there was a funneling step similar to when I built the index. Depending on if I wanted to search using stems, lemmas, or neither (plain), it will initialize the appropriate QueryEngine object and use the corresponding Analyzer (Standard or Whitespace) and direct it to the right index. Again, for this naïve approach I did not filter any stop words from the query, but I did delete the sequence `(Alex:` from the clue, which indicates a comment from the host Alex Trebek and does not actually contribute to the answer. Including `Alex` in the clue might negatively affect the results.

3 Measuring Performance

In this section I will discuss the initial results using my naïve retrieval approach. In the *Jeopardy!* gameshow, the goal is to get the unique right answer to the clue. Because of this, my measure of performance depends only on the top 1 result; if the correct answer is close to the top but not the right choice, it is considered incorrect. To capture this I use precision at 1 to measure the performance of my system. To calculate this, I simply take the number of correct results divided by the total number of questions (100 in this case). results for this initial retrieval model are given in Table 1. So far, we see that stemming performs the best, followed by lemmatization.

Index	P@1
Stems	**0.18**
Lemmas	0.16
Plain	0.15

Table 1: Precision at 1 scores for the naïve model with default BM25 scoring ($k_1 = 1.2$, $b = 0.75$).

4 Changing the Scoring Function

Perhaps one reason for the lower scores is the scoring function. By default, Lucene uses BM25 (previously it used a vector-space scoring function based on *tf-idf* weighting). The two important parameters in BM25 equation are k_1, which normalizes term frequency, and b, which normalizes document length. Lucene's default sets $k_1 = 1.2$ and $b = 0.75$.

Considering many of the Wikipedia pages provided here (and in general) are extremely short, sometimes only being redirects to other pages, I experimented with changing the value of b to change the scoring function. Setting $b = 0.1$ yielded improved results compared to the naïve approach, as shown in Table 2.

Index	P@1
Stems	**0.39**
Lemmas	0.21
Plain	0.21

Table 2: Precision at 1 scores for the naïve model with modified BM25 scoring ($k_1 = 1.2$, $b = 0.1$).

By setting $b = 0.1$, I penalize longer pages less. Considering the short Wikipedia pages are often stubs, it seems more likely that a longer page will be more relevant for a *Jeopardy!* answer than a shorter page. This is just a hypothesis for now.

5 Error Analysis

In this section, I will describe two classes of errors that I found from the naïve approach, discuss ways possible ways to address them, and compare results so far between my three indexes.

One class of errors is shown in Figure 1. Figure 1 shows the query, which includes the clue and the category, the correct answer, and the top 10 hits using the modified BM25 similarity function for two different *Jeopardy!* questions. We see in both examples that the correct answer is in the top 10 hits (number 3 and number 2, respectively), so with some modification we might be able to promote it to the top hit. In addition, the top hit includes one or more words present in the clue itself. This is unlikely to be the case in a real *Jeopardy!* question; the point is to guess the right an-

swer, so including the answer in the question does not make sense in the context of the game.

One way we could address this class of errors and promote the correct answer is to take the context of the game into consideration. If the top hit contains too many words that overlap with the clue, move on to the next hit until you reach one that does not overlap with the clue.

A second class of errors is shown in Figure 2. In these examples, the words in the clue and category are not informative enough as a bag of words to get the right answer. In this case, the correct answer is not found in the top ten hits.

One way to address this is to use phrase queries. In these clues, the song titles are provided in double quotes. Taken as a bag of words, the words `rock`, `with`, and `you`, together with the rest of the query, don't really help identify the song. However, if we treat "Rock With You" as a phrase query and enforce that the words show up next to each other, the correct answer will more likely show up near the top.

Finally, the results in Table 1 and Table 2 show that stemming performs better than either lemmatization or doing neither stemming nor lemmatization. The latter two methods perform similarly for both conditions, and while stemming only performs 2–3 points better with default BM25, the difference jumps up to 18 points when using my modified BM25 scoring. The reason doing nothing performs worse seems obvious; by not normalizing in any way, a close match that differs slightly in form will not count as a match.

Why lemmatization performs similarly, or even worse than "plain" with modified similarity, is somewhat less clear. One possibility is that I used Lucene's WhitespaceAnalyzer both for indexing and for processing the query. For the index, this was not a problem, since the Wikipedia page text was tokenized and lemmatized before being joined with spaces. However, the query was not processed in the same way. The query words were simply split on spaces, without being tokenized first. This can be a problem for cases like `church,`, where if tokenized, the `,` will be separated from `church`. If not tokenized first, the whole string `church,` will be sent to Lucene as a unit and might end up not matching the string `church`.[1]

6 Improving Retrieval

In this section, I describe my attempt to improve the retrieval system by addressing the first of the error categories discussed above.

As mentioned before, one way to address the problem of the answer containing words from the query is to simply skip over results that contain too many words from the query. I implemented this in two ways. First, I filter the query to only include words with a subset of part-of-speech tags. This is to eliminate meaningless overlap, such as if the answer and the query both had the word *to* or shared punctuation. The tags I use include all tags for nouns, verbs, and cardinal numbers. I experimented with other tags such as adjectives and adverbs, but adding those did not change the performance of the system.

After filtering the query, while comparing returned results with the correct answer, the model will keep track of how many overlapping words there are between the correct answer and the top hit. For my best model (reported here), the overlap had to be less than one. That is, if the top hit and the answer shared even one word (with the relevant tag), the hit was rejected. Then, the system considers the next hit on the list in the same way, and so on until the current 'top' hit does not overlap with the answer.

This approach touches on an important linguistic principle (in addition to making sense for *Jeopardy!*). Mainly, certain words are important for carrying the central meaning of a sentence, while others are not. These (typically function) words are called stop words. For example, the word *the* in a query is likely to be uninformative, since it occurs in practically every document. When originally processing the Wikipedia documents for building the indexes and when building the queries, I did not filter out stop words. Despite ignoring this principle originally, post-hoc filtering based on part-of-speech tag ultimately resulted in my best-performing system.

Results of my improved model, using both the default BM25 scoring function and my modified

[1] I encountered this with my improved model, where `church` was not considered to overlap with `church,`, yielding a wrong answer for the top hit.

Index	w/ Default Similarity	w/ Modified Similarity
Stems	**0.25**	**0.43**
Lemmas	0.19	0.32
Plain	0.21	0.34

Table 3: Precision at 1 scores for the improved system for the stems, lemmas, and plain indexes, comparing the default BM25 similarity function with my modified BM25 similarity function.

BM25 scoring function, are given in Table 3.

7 Conclusion

Overall, the best retrieval system uses an index based on stemming (with Lucene's StandardAnalyzer), a modified BM25 scoring function where $b = 0.1$, and post-hoc filtering of the query based on part-of-speech tags and skipping hits that have too much overlap with the original query.

There are other clear ways to try to improve this system, including introducing phrase queries to address the second error class I identified, or removing stop words from the original index. I reserve these for future work.

```
Query:  Not to be confused with karma, krama is a popular accessory sold in
cambodia; the word means "scarf" in this national language of Cambodia CAMBODIAN
HISTORY & CULTURE

Correct answer: khmer language

Using modified BM25 Similarity:
Hit 1:  DocName: History of Cambodia    DocScore: 59.53534698486328
Hit 2:  DocName: Politics of Cambodia   DocScore: 59.104557037353516
Hit 3:  DocName: Khmer language DocScore: 58.31053161621094
Hit 4:  DocName: Khmer Rouge    DocScore: 57.4572868347168
Hit 5:  DocName: Economy of Cambodia    DocScore: 51.73160171508789
Hit 6:  DocName: Siem Reap      DocScore: 48.53266525268555
Hit 7:  DocName: Ho Chi Minh City       DocScore: 48.411109924316406
Hit 8:  DocName: Ikat   DocScore: 47.433082580566406
Hit 9:  DocName: Paan   DocScore: 46.23930358886719
Hit 10: DocName: Mekong DocScore: 46.0582275390625

Query:  Don Knotts took over from Norman Fell as the resident landlord
on this sitcom THE RESIDENTS

Correct answer: three's company

Using modified BM25 Similarity:
Hit 1:  DocName: Don Knotts     DocScore: 61.50598907470703
Hit 2:  DocName: Three's Company        DocScore: 44.23488235473633
Hit 3:  DocName: Tim Conway     DocScore: 35.788169860839844
Hit 4:  DocName: John Ritter    DocScore: 33.25556564331055
Hit 5:  DocName: Bill Bixby     DocScore: 30.387065887451172
Hit 6:  DocName: Matlock (TV series)    DocScore: 30.287059783935547
Hit 7:  DocName: San Pedro, Laguna      DocScore: 29.240985870361328
Hit 8:  DocName: That '70s Show DocScore: 28.738615036010742
Hit 9:  DocName: Malta  DocScore: 28.33993148803711
Hit 10: DocName: County Mayo    DocScore: 28.2568359375
```

Figure 1: Examples where the highest-rated answer contains a word or words present in the clue. *Jeopardy!* answers are unlikely to contain words that are in the clue (excepting punny categories), so skipping results that *do* contain overlapping words might fix these errors.

```
Query:  1980: "Rock With You" ’80s NO.1 HITMAKERS
Correct answer: michael jackson

Using modified BM25 Similarity:
Hit 1:  DocName: Orchestral Manoeuvres in the Dark  DocScore: 27.115528106689453
Hit 2:  DocName: Cold Chisel    DocScore: 25.40660858154297
Hit 3:  DocName: Wolfman Jack   DocScore: 25.255949020385742
Hit 4:  DocName: The Runaways   DocScore: 25.107824325561523
Hit 5:  DocName: Helix (band)   DocScore: 24.45211410522461
Hit 6:  DocName: Whitesnake     DocScore: 24.35768699645996
Hit 7:  DocName: Duran Duran    DocScore: 24.109310150146484
Hit 8:  DocName: Laibach (band) DocScore: 24.034826278686523
Hit 9:  DocName: Glam metal     DocScore: 23.971431732177734
Hit 10:         DocName: The Moody Blues        DocScore: 23.339218139648438

Query:  1989: "Miss You Much" ’80s NO.1 HITMAKERS
Correct answer: janet jackson

Using modified BM25 Similarity:
Hit 1:  DocName: Helix (band)   DocScore: 27.506011962890625
Hit 2:  DocName: Neneh Cherry   DocScore: 25.372238159179688
Hit 3:  DocName: George Michael DocScore: 25.11778450012207
Hit 4:  DocName: I Love the ’80s Strikes Back   DocScore: 25.046329498291016
Hit 5:  DocName: Book of Love (band)    DocScore: 23.877687454223633
Hit 6:  DocName: Gloria Estefan DocScore: 23.783161163330078
Hit 7:  DocName: Halle Berry    DocScore: 23.55036163330078
Hit 8:  DocName: Annie (musical)        DocScore: 22.938392639160156
Hit 9:  DocName: Orchestral Manoeuvres in the Dark  DocScore: 22.803762435913086
Hit 10: DocName: NBC    DocScore: 22.757150650024414
```

Figure 2: Examples of where the words in the clue and category are not informative enough to get the right answer. Could possible be solved with phrase queries using the song title given in the clue.