Ben Bauer

CSC 483 Fall 2020

CSC 483 Programming Project Analysis : Building (a part of) Watson

Link to GitHub repository: https://github.com/LennonWalrus/483FinalProject

Link to Lucene Index:
https://drive.google.com/file/d/1exodfuBBlak50CnitYY36etvX3zY1EAd/view?usp=sharing

Instructions to Run:

1. Clone Github Repository

2. Download and extract Lucene Index.

3. Place wiki.lucene folder in src/main/resources directory of GitHub project.

4. On command line move to the project directory.

5. Compile using command "mvn clean install" (may take a minute or so)

6. run using either "mvn exec:java -Dexec.mainClass=Query" for Top 1 document scored or mvn exec:java -Dexec.mainClass=Query -Dexec.args="x" where x is the top x documents scored.

Note: To run other branches(configurations) you will be required to put the wiki text files in a folder named "WikiPages" and also insert that into src/main/resources.

Code Description:

Index.java is the file that goes through and indexes the nearly 280,000 Wikipedia articles found in the provided text files. It starts by creating a FSDirectory to store the Lucene Index at src/main/resources . Then it finds the Wikipages directory holding the 80 text files of wiki pages in the resources folder and starts looping through each text file. Each file is scanned line for line to detect wiki titles which are cleaned using LineCleanerT and lemmatize each pages contents using NLP simple api lemmatization. During this process lines are passed to the LineCleaner method to remove unnecessary punctuation from tokens and eliminate some of the formatting of wiki pages. Once a wiki page is done being indexed it is added to the lucene index using its title and context.

Query.java is that file that answers the 100 questions provided in the questions.txt using the lucene index built from Index.java. It has an optional argument that dictates how many top documents are scored with no argument defaulting to the top 1 document. The program starts by using the grabIndex function to find the index generated by Index.java and set it for use. Then the program goes through the questions.txt and grabs the hints and answers for the questions. It then takes the hints (category and question) and lemmatizes them the same way the pages were lemmatized in the index using the queryLemm method. Then it takes the lemmatized hints and builds lucene queries in the luceneAnswers method. Here it builds the queries by adding "OR"'s between each token and then generates the Lucene query with the resulting string. Then a reader is built using the index and the searcher is built using the reader, query, and using the standard Bm25 similarity. The top documents are compiled into a ScoreDoc array of which the top 1 or X documents are checked for a match with the correct answer saved from earlier. The MRR is computed using the docMRRCalc method which checks for how far down the list the correct match is and computes the MMR score appropriately. For each question, the lemmatized hints, the top 1 or x matches, and the answer are printed, and then the final MRR score is printed once all questions have been processed.

Question 1 should be addressed throughout Index.java and Query.java. Question 2 has its codebase in the LuceneAnswers Method and DocMRRCalc method in Query.java. Question 3 code is found in the TFIDF Branch, not the master branch. Question 4 has code in all the branches as that's how the configurations are spread out. The query-specific answers can be understood by looking at the output of the program and cross-referencing the scores for each question with the queries used for them.

Written Question Responses

1.) The Wikipedia content required a fair bit of alteration for the terms to be properly tokenized, lemmatized, and then indexed and used for querying. In my final implementation I had a small amount of stemming (mostly in removing these characters {(),?.!;:="} from the ends and middle of the tokens) and then using Core NLP's Simple API to lemmatize the tokens and then index the lemmatized sentences. The Wikipedia pages themselves had some formatting that needed to be addressed in order to index the pages given. Each Wikipedia title had double brackets around its name, but so did every file and image, so I had to ensure that what was inside the double brackets wasn't a file or image to properly index each page as a document. I did not index several things that were in the wiki articles like the often repeated "#redirect", "CATEGORIES:" statements. I also choose to remove the text within [tpl][/tpl] sections as they were describing other sources for something already in the wiki articles themselves as well.

    As for how I built the queries, I used both the category and clue lines to form my queries. I used all the words in the category and clue lines except for those found in Lucene's stop analyzer ENGLISH_STOP_WORDS_SET (though stop words did not seem to affect the outcome see results below), which is what I used to filter out the stop words in the query, and then used Lucene's "OR" in the query to tie the terms together. From my testing, it was almost always beneficial to include the category in the query as it would raise the number of questions answered correctly (except in the case of TDIDF).

2.) To measure the performance of my system, I decided to use the mean reciprocal rank (MRR). I decided to go with MRR because of the importance placed on the top document being found being emphasized as this is the answer our Watson would be presenting. With MRR, I could get a good understanding of one: the rate of the actual top document being returned if the top doc is set to just 1 and two: how close the actual document was to being on the top of the ranking when top docs was set to anything more than 1.

3.) I replaced the scoring systems by swapping out Lucene's default similarity (Bm25Similarity) with its classic similarity (TF-IDF) for the searcher in my Query.java and index config in my Index.java and found that the Classic Similarity performed much worse than the base BM25 similarity did. The classic similarity would only be able to answer 1 question right given the many configurations I tried with it whilst the base similarity could answer 22 with lemmatization. (See results at the end of the document). I also found it strange that I got the best MRR score using raw text instead of the lemmatized text for the indexing.

4.) My best system was able to answer 22 questions correctly and 78 incorrectly.

Correct Queries

I believe that the questions that were answered correctly were done so because many of those queries contain either a unique name or word or a set of them that were not common amongst the rest of the wiki pages. For example "McGivney" for Knights of Columbus, "otter flounder Pinto & Bluto" for Animal House, "Malia" for Michelle Obama, "heptathlon" for Jackie Joyner-Kersee, "Daniel Hertzberg & James B Stewart" for The wall street journal" and "McCain-Palin" and "Julianne Moore" for Game Change.

Another reason that some of these might have been correct even with the simplicity of the system was a specific date and location combination in some of the queries that could really work to narrow down perspective documents. For example "New Orleans" and "Sept 25 2006" for Mercedes-Benz Superdome or "May 5 1878" and "Dodge city kansas Cemetery" for Boot Hill.

Lastly, it seemed to answer all the questions about Greek food, though the only reason I can think for this is that the queries describe the process of how the foods are made and those descriptions have a few distinct terms that in combination may make finding the correct page easier like cure, store, brine, pickle, and crumbly for feta or “marinate lamb skewer & grill” for souvlaki.

Incorrect Queries

As for the questions that my configuration could not answer there are several categories. Though the configuration did manage to correctly answer several queries on people, it did not do so well on the whole and especially with those identified by their works or quoted words. This might be the case because of the scant information provided in these queries (besides the quoted words or works) and how the individual terms work better together than separate. One example could be the lemmatized “golden globe winner 2010 sherlock holme film” for Robert Downey Jr, where “golden globe winner” together may have a lot of meaning as well as “sherlock holme film”, but when used piecemeal can obfuscate the original query. For similar lemmatized queries see “'80 1 hitmaker 1980 rock you” for Michael Jackson where “rock with you” the original song name is not only cut short due to lemmatization but “rock” and “you” are somewhat general words that may be found in several thousand articles. Other categories such as “Poets & Poetry”, “Broadway Lyrics” and “Historical Quotes” quote either words from the work or the titles themselves which get split up and lemmatized like “Rock with you” did for Michael Jackson and to much of the same effect like the lemmatized “poet & poetry she write my candle burn both end ah my foe oh my friendsit give lovely light "” for Edna St. Vincent Millay which is a bundle of words that can be found across many different articles, but if taken in the exact combination would be much more powerful in a search.

Some of the queries and their categories are written in a way that a simple bag of words implementation may falter. For example all of the questions categorized as “STATE OF THE ART MUSEUM (Alex: We'll give you the museum. You give us the state.)” if not using the category, will only have names of the museum like “The Georgia O'Keeffe Museum” to go off of which will not point to a big article like “New Mexico” which is the answer. Even given the category, words in the category field like art, museum, and Alex will outweigh the scoring system efforts to find a state even though “state” is added to the bag of words as well. Similar to these are the queries categorized as “CAPITAL CITY CHURCHES (Alex: We'll give you the church. You tell us the capital city in which it is located.)” for much of the same reasons.

Other categories like “1920s News Flash!” are stylized in such a way that they add several unnecessary words that also throw off the bag of words approach. Such as “less-than-yappy pappy” and “top dog after chief takes deep sleep” for Calvin Coolidge or “this empire, infact!” and “it’s goodbye, this, hello” for Ottoman Empire. These inclusions require logical leaps by humans that cannot be replicated by a simple system like this.

Implementation Impacts on System

My best configuration by MRR scoring both for top document found and for the top 10 documents found for the system ended up being to use lemmatization with the use of the categories provided. This is likely due to more relative words within the queries matching the index words due to the lemmatization process over my stemming implementation. Only when the exact spelling of words match will they form a match in the system, so when we lemmatize text both in the query and index, we are trying to maximize words with similar meanings matching up by changing the words outright to lemmas that share the same meaning and coincidently the same spelling. However, the performance when not using the categories dipped slightly below that of stemming for the top 1,3 and 5 documents returned but again surpasses it if the top ten documents are used. It should be noted that the top document found is most important as this is what would be used and returned by Watson.

See Results on Next Page

Results

MRR scoring per configuration using the 100 questions

(When Top Doc# = 1 every .1 is a correct answer found)

Top Doc#	Lemma/no category/no stopword filter	Lemma/category/no stopword filter	Lemma/no category/stopword filter	Lemma/category/stopword filter
1	.17	.22	.17	.22
3	.21833333	.26333335	.21833333	.26333335
5	.23783335	.27883336	.24166667	.27883336
10	.24787302	.28602383	.24787302	.28602383

Top Doc#	Stemming/no category/no stopword filter	Stemming /category/no stopword filter	Stemming /no category/stopword filter	Stemming/ category/ stopword filter
11	.18	.20	.18	.20
3	.2366667	.24166667	.2366667	.24166667
5	.2516667	.2591667	.2516667	.2591667
10	.25659525	.2719445	.25659525	.2719445

Top Doc#	No Stem/no category/no stopword filter	No Stem / category/ no stopword filter	No Stem /no category/stopword filter	No Stem/category /stopword filter
1	.14	0.18	.14	.18
3	.18500002	.21166667	.18500002	.21166667
5	.19199999	.21416669	.19199999	.21416669
10	.20049208	.22977382	.20049208	.22977382

Top Doc#	TF-IDF /no category/no stopword filter	TF-IDF /category/no stopword filter	TF-IDF /no category/stopword filter	TF-IDF /category/stopword filter
1	.01	.01	.01	.01
3	.013333334	.01	.013333334	.01
5	.013333334	.012	.013333334	.012
10	.015	.013111111	.015	.013111111