David Minch
CSc 483
5/1/19

BUILDING A PART OF WATSON: WATSON VERSION 0.2!

Welcome to WatsonMinch, the best Jeopardy Competitor this side of the... um... Okay, maybe it's not the best, but I was really happy to bring the success up to 15%.

Table of Contents:

Github Repository Link

https://github.com/minchdavidm/CSc483WatsonMinch.git

Include a description of the command line to run the code, with an example.

WatsonMinch is built on Maven, using Java, and the Java Libraries Lucene and StanfordNLP. The pom.xml file will fetch all of the required files.

The program includes functionality for some different flags. I'll describe and give the default behavior first, as that's most likely the function you want to run. Note that all of these commands have accompanying shell scripts that can be run, that way you don't have to copy and paste the command.

Default Behavior: WatsonMinch looks for the premade Lucene Index archive, loads it in, reads in the Jeopardy Clues file (src/main/resources/questions.txt), lemmenizes and tokenizes the Clues file, then for each clue, WatsonMinch scores documents using the Lucene Index archive, and stores the top scored document. Then, for each Jeopardy clue, it's printed out Jeopardy Style.

Instructions to run with default behavior:

```
mvn compile
mvn exec:java -Dexec.mainClass="CSc483.WatsonMinch"
```

Or run the shell script `runDefaultBehavior.sh`, which will run these same two commands.

The results of the default behavior will print out the clue and the response in the format:

```
And the answer is:
> This Georgia paper is known as the AJC for short
What is...  The Atlanta Journal-Constitution?
That's right!
```

Or, if the answer is wrong, it'll display the correct answers, as so:

```
And the answer is:
> Bessie Coleman, the first black woman licensed as a pilot, landed a street named
in her honor at this Chicago airport
What is...  Bessie Coleman?
That's wrong.  Possible responses:
O'Hare
O'Hare International Airport
```

David Minch
CSc 483
5/1/19

Optional Behavior: Provide own query

```
mvn compile
mvn exec:java -Dexec.mainClass="CSc483.WatsonMinch" -Dexec.args="-q Query Terms Go Here"
```

Alternatively, the shell script `runWithQuery.sh Query Terms Go Here`.

This will lemmenize and tokenize just the given clue, and print the tokenized query together with the top 10 scoring documents and their scores.

Example for query "This pokemon is yellow, electric, and ashs best friend."

```
Original query:  This pokemon is yellow, electric, and ashs best friend.
Tokenized query:  thi pokemon yellow electr ash best friend
Potential Question:  Raichu, with score 34.205726623535156
Potential Question:  Pikachu, with score 32.936424255371094
Potential Question:  Squirtle, with score 31.558313369750977
Potential Question:  Charmander, with score 30.063858032226562
Potential Question:  Charmeleon, with score 27.9473819732666
Potential Question:  Venusaur, with score 23.866960525512695
Potential Question:  Blastoise, with score 21.138690948486328
Potential Question:  You're My Best Friend (Queen song), with score 19.805192947387695
Potential Question:  Jigglypuff, with score 19.46177101135254
Potential Question:  Mewtwo, with score 18.723318099975586
And the answer is:
> This pokemon is yellow, electric, and ashs best friend.
What is...  Raichu?
```

Optional Behavior: Request reparsing of the Wikipedia Data

```
mvn compile
mvn exec:java -Dexec.mainClass="CSc483.WatsonMinch" -Dexec.args="-p"
```

This one I didn't provide a shell script for because it will take hours to run and you certainly don't have that kind of time while grading. Just know that giving it this flag does the parsing instead of reading in the Lucene Index, then the rest of the program is the same. You can mix flags, for example, give the -p flag then the -q Query Terms flag, and it will parse the wikipedia files, and at the end, run the given query instead of the Questions.txt file.

Optional Behavior: Verbose output

```
mvn compile
mvn exec:java -Dexec.mainClass="CSc483.WatsonMinch" -Dexec.args="-v"
```

This one I didn't provide a script for either because 1) Running the -q flag automatically turns on verbose output, and 2) It's way to much information to handle on the screen with all 100 queries.

This flag tells the program to list the 10 best potential questions with their scores before providing the normal Jeopardy format answer and question.

Provide a description of the code.

WatsonMinch.java This file handles the main class for the program. After checking for parameters as described above, it does the following:

1) Gets the Lucene Index

1a) If the program is using default behavior, it checks the `src/main/resources/lucene-files/` directory for a premade Lucene Index and loads it in as an IndexReader.

1b) If the program is given the -p flag, it calls the WikipediaParser class, which checks the `src/main/resources/wi` directory for the wikipedia files, parses them, lemmenizes them, feeds them to Lucene, and returns the IndexReader.

2) Gets the Queries

2a) If the program is using default behavior, it checks the `src/main/resources/` directory for an archive named `questions.txt`, and loads in the queries, or fails with an error if the file is not properly formatted.

2b) If the program is given the -q flag, all terms after the -q flag in args are given as terms for the query.

3) The TextLemminizer is called to lemmenize the queries, returning a HashMap mapping the queries to their lemmenized versions.

4) The WikipediaParser is called to score the lemmenized queries, returning a HashMap mapping the lemmenized queries to an array list of ScoredDocument objects, each of which holds the document ID and the score that document got for that query.

5) Verbose output is then given if requested.

6) The remaining details from questions.txt, including the correct answers are loaded in if a query was not given.

7) Then for each query, the top document is returned as a Question for that Jeopardy Answer Clue.

7a) If we're using the questions.txt file, it also reports if the Question is right, and, if wrong, what were the possible Questions that could be given as a response.

8) If we're using the questions file, the effectiveness of WatsonMinch is evaluated using the TODO approach.

WikipediaParser.java This file handles the functions for parsing the Wikipedia files, and for scoring queries based on the generated index.

1) parse()

1a) The StandardAnalyzer, FSDirectory, IndexWriterConfig, and IndexWriter are all instantiated.

1b) The list of Wikipedia files in the directory is retrieved.

1c) A Scanner is used to read the files line by line.

1c i) Check the line to see if it's a valid page title

1c ii) If it is, then check if we were already reading information for a page

1c iii) If we were, store the previous information with it's page title as a new document

1c iv) To store it, lemmenize the document text first by StanfordNLP. ← This raised my accuracy from 13 to 15 over just having Lucene tokenize the text itself.

1c v) If the page was not a valid page title, it was body text, so store it for the next iteration.

1d) After all files have been read in and stored in the IndexWriter, the changes are committed.

1e) The IndexReader luceneIndex is opened on the same index.

2) score(HashMap<String, String>)

2a) A Query ⇒ Query Parser object is built to process the queries.

2b) The Query Parser parses each lemmenized query

2c) A list of 10 ScoredDocument objects is created

2d) An IndexSearcher object is instantiated over the luceneIndex generated by parse()

2e) The 10 best scoring documents are stored with their document ID and score for this query.

2f) The overall results are returned as a hashmap mapping the lemmenized query to it's list of 10 best documents.

TextLemminizer.java This class contains static methods that stems, lemmenizes, tokenizes and simplifies a given String based on the following set of rules: 1) The default Stanford CoreNLP Tokenizing and Lemmenizing rules 2) Punctuation is then removed 3) Stop words are removed 3a) If it's a query, then if the query contains more than 4 words, stop words are removed, that is, short queries are treated like quotes 3b) If it's a document, then any stop words that aren't part of a quote are removed
1) The global booleans STEM and LEMMENIZE are set. I got the best results by having both set to true for both queries and building the index.
2) Given a string s:
3) The properties for a StandfordCoreNLP pipeline are set up
    3a) If LEMMENIZE is set to true, the properties are ("annotators", "tokenize, ssplit, pos, lemma")
    3b) Otherwise, the properties are just ("annotators", "tokenize, ssplit, pos").
4) An Annotation is created with the string s, and the pipline annotates it.
5) Sentence by sentence, the terms only containing punctuation are removed
6) An array of tokens collecting all of the tokenized terms is made to hold the results
    6a) Two compound if statments are used to decide what combination of LEMMENIZE and STEM we're doing.
    6b) I got the best results by manually downloading Stemmer.java from the StanfordCoreNLP github repository, and calling that for stemming after Lemmenizing. I tried both before lemmenizing (that just ruined it), and I tried relying on Lucene for the stemming, but the StanfordCoreNLP Stemmer gave me somewhat better results, so that's the one used in the final submission.
7) Stop words are removed from the string s
    7a) Stop words are removed just by checking a list of common stop words.
    8) The tokenized string is returned

ScoredDocument.java This class is just a wrapper to join the Document ID and Score. It's more like a C struct than an actual class.

Answers to the parts of Question 1:

Describe how you prepared the terms for indexing (stemming, lemmatization, stop words, etc.)

I ended up getting the best results by doing stemming, lemmatization, and removing stop words on both the documents while parsing, and on the queries.

I built my code around some boolean flags that could turn on and off stemming and lemmatization, built several Lucene indices with each, and then tested them with queries that were also stemmed and lemmatized optionally. As could be expected, the best results came from having both flags set to true.

How I Lemmatized the strings: On the previous page, I describe the workings of my TextLemminizer.java file. The basic rundown of the lemmatization part is that I would load the strings line by line from the Wikipedia dumps, lemmatize them with the properties ("annotators", "tokenize, ssplit, pos, lemma"), then manually remove the punctuation from the resulting lines. Then the tokens would be joined for the wikipedia index.

How I Stemmed the strings: I did a couple of tests, and found that most strings that I sent to Lucene were staying pretty much the same. They would get the 's' lopped off plurals, but not much additional stemming was going on. Therefore I looked to see if the StanfordCoreNLP had any Stemming tools. It did, but not in the basic package. The instructions were to simply download the file from Github and include it in the classes, as there wasn't a standardized way to get them via maven. As such, I downloaded Stemmer.java from the StanfordCoreNLP github page, removed a few methods that were codependent with other files such as the Main method, and once I did that, I made a Stemmer object in my Text Lemmenizer class, and used the stemmed versions to put into the Lucene text body part of the index. I worried that this might hurt my performance because things would potentially stem twice, as I didn't look into how to turn off Lucene's stemming, but when I reindexed the Wikipedia files, I had two more answers correct that I didn't have before, so I was happy.

How I removed stop words: I got a list of stopwords from `http://xpo6.com/list-of-english-stop-words/`, and just removed all occurences of these words from all documents' text. However, I did something a little different for the query: As some queries were short, I decided to let the articles be indexed in those cases so that if there were any quotes of just a few words, the articles in the query's quote would help us find those specific documents.

What issues specific to Wikipedia content did you discover, and how did you address them?

As I asked initially on Piazza, one of my biggest worries was how to be sure what I was looking at was a title. The way I got around this was ultimately by considering a line a title if it starts with "[[", ends with "]]", and does not contain any pipe characters. I verified that that didn't eliminate any invalid titles on 4 of the wikipedia dump files (tests with grep), and was ultimately satisfied with that option.

Another issue was the sheer amount of content to index. I was running into memory errors when trying to parse entire documents at once with the StanfordCoreNLP. I got around this by parsing the files myself using the scanner, and then just giving one line at a time to the StanfordCoreNLP, and joining the strings myself before passing them to Lucene. Once I did that, it took longer to run (2 hrs before, 4.5 hours after making the change), but I was able to do 4 builds since then without any more memory errors, so I think it was a good decision to make that change.

Describe how you built the query set from the clue. For example, are you using all the words in the clue, or a subset? If the latter, what is the best algorithm for selecting a subset of words in the clue? Are you using the category of the question?

The query set is pretty close to the clue. I run the query set through the TextLemmenizer class, which, as described above, has some rules about how it makes a subset out of the query. If the query is longer than 4 words, it removes stop words until there are none, or the resulting query has 4 words. That way, things in quotes that weren't lemmenized in the documents might be approximated if the query is really short.

As for the category of the question, I first implemented the index without care for the category. Once I had things working, I added on the category as another value in the Lucene index, and that actually made my accuracy **worse**. I think this was because my Lucene index was giving too much weight to the category, but since the Wikipedia documents didn't have a category value to compare to, it was just treating it like a super important token compared to the other things, and was instead giving me documents that had to do more with the category word than the query. Though I had one more idea for how to implement the category, and that was to tack on the category as if it were another term in the query. That way, it wouldn't get the super weight of being its own field, but it could still help get rid of some of the worst choices. When I did this, I managed to get up from my previous best (15 out of 100 queries right) to 21 queries right. I did not attempt to evaluate if I was getting 6 more right, or if I was getting more than 6 right and some other ones wrong that I was previously getting right.

So all in all, my best algorithm was: Add category as a first term in the query, Lemmenize, Stem, Remove stop words (with rules defined above). That's what gave my current best of 21.

David Minch
CSc 483
5/1/19

2. Measuring Performance

Measure the performance of your Jeopardy system, using one of the metrics discussed in class, e.g., precision at 1 (P@1), normalized discounted cumulative gain (NDCG), or mean riciprocal rank (MRR). Justify your choice, and then report performance using the metric of your choice.

I chose to use the Mean Riciprocal Rank (MRR) for measuring my queries. There are 4 reasons why I chose this:

1) We're fetching the top 10 documents, ranked 1-10. So we have all the information needed to evaluate the MRR.

2) Some systems aren't too critical if you get a document ranked close to 1 rather than exactly 1, however for Jeopardy answers, you get 1 guess at the question, so it's essential to get as close to 1st place as possible. So raising MRR has a great return even if we don't raise the number of documents who get one in the top 10.

3) In my opinion, it's the easiest metric to get right, along with the one I best understand. Maybe that influenced my decision a little, but I still stand by it as my choice.

4) This metric doubles as a test statistic. This probably applies basically for all useful metrics, but there exists the advantage that I could do further statistical analysis on this data set (if I assume the selection of queries to be random, which it really isn't because they come from several set categories, so maybe I'd want another data set, but the point applies) to come up with an estimate of how effective WatsonMinch is if it were to enter a Jeopardy competition. I'm in Math 466 right now, so that kind of application is at the top of my mind and most calls my attention.

So now let's calcuate it!

$MMR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}$. In this case, Q is the set of queries, so $|Q| = 100$, and here are my results:

List of result numbers where the document wasn't in 1st place:
4, 8, 5, 8, 3, 5, 2, 3, 4, 5, 5, 2, 8, 9, 2, 9, 2, 5, 3, 4, 6, 2, 3, ← Totalling 23 documents that had the correct query in the top 10 documents but not the first. These results are reported to verboseResults.txt in the mvn source directory. Note when viewing those results, you can scan finding the word "However," and only the lines reporting the document result numbers will appear. Also note that these are 0-indexed numbers, so each of the values added here are 1-indexed, so add 1 to recreate my results.
21 out of 100 were spot on.
For the remaining 56 documents that did not make the top 10, we're going to assign a pessimistic value of 0 for their *riciprocal* rank. Note that means $\frac{1}{\text{rank}_i} = 0$, not that $\text{rank}_i = 0$. That way, for further studies, getting a document into the top 10 is worth way more than a change from maybe 50 to 15, because, to put it simply, if it's not in the top 10, it's not worth spending time on until we get all the documents from the top 10 to the pole position. Thus our sum is: $\frac{1}{100} * (21 * \frac{1}{1} + 5 * \frac{1}{2} + 4 * \frac{1}{3} + 3 * \frac{1}{4} + 5 * \frac{1}{5} + 1 * \frac{1}{6} + 3 * \frac{1}{8} + 2 * \frac{1}{9} + 56 * 0) = (21 + 5/2 + 4/3 + 3/4 + 1 + 1/6 + 3/8 + 2/9 + 0)/100 = .273472$.

I believe that this is a good value to represent my data: It's still very close to the 21% accuracy my documents experienced if they were playing Jeopardy, but it does represent that some answers were close, though it's no where near the 44% of documents that were in the top 10 options. If , with my changes that I'll mention in the Error analysis, could bring the MRR above 40%, I would call my solver very successful.

3. Changing the scoring function

Replace the scoring function in your system with another. For example, by default Lucene uses a vector-space scoring function based on $tf \cdot idf$ weighting. You can replace this default choice with a probabilistic scoring function, such as BM25. How does this change impact the performance of your system?

I decided to implement BM25 as suggested in the question. The BM25 formula is:
$\text{RSV}_d = \sum_{t \in q} \log\left[\frac{N}{\text{df}_t}\right] \cdot \frac{(k_1+1)\text{tf}_{t,d}}{k_1((1-b)+b\times(L_d/L_{ave}))+\text{tf}_{t,d}}$.

That is, sum the log of the number of terms in the document divided by the document frequency of that term, times a ratio involving k_1, which is the tuning parameter for document term frequency scaling, L_d, which is the length of document d, divided by L_{ave}, the average length of documents in the collection, and b, the tuning parameter controlling the scaling by document length.

This was a little hard to implement until I realized that Lucene's CollectionsStatistics (accessed through the IndexSearcher that I was already using) lets us get the sumTotalTermFreq() and maxDoc() so that I could get L_{avg}. This allowed me to make the (L_d/L_{ave}) ratio for each document. I then chose 25 for k_1 and 1 for b.

Once I implemented the calculation, I rescored the queries. The overall results were fairly similar (Previously with this setup (ignoring the categories), I was getting 15 right, and with this, 14 right. However, I reviewed some of the ones I got wrong, and found an interesting pattern:

For the query "In 2010: As Sherlock Holmes on film", the top scored document was List of actors who have played Sherlock Holmes. ← This result was a page with just a list of names, each of which was followed with an "As Sherlock Holmes".

For the query "Italian for "leader", it was especially applied to Benito Mussolini", the top scored document was "Fascist and anti-Fascist violence in Italy (191926)", which was a redirect page with almost no content.

For the query "The dominant paper in our nation's capital, it's among the top 10 U.S. papers in circulation", the top result was Media of the United States, which is a shorter page with links to longer articles on each subject.

For the query ""Patriot Games"; he's had other iconic roles, in space & underground", which resulted in Twin Series Vol. 3 Konchuu Monster/Super Chinese Labyrinth, which is again a very short redirect page.

From this small sample of results, I concluded that the BM25 was giving too much bias towards the shorter documents, which, as I didn't remove redirects or lists from the queries, was overly favoring short articles that had a few terms in common compared to long documents with all the terms.

Therefore, I restored the original functionality for Lucene, although I should have planned more time to change those parameters and toy with the data some more to see if by adjusting the tuning parameters, I could have outdone Lucene's default behavior.

David Minch
CSc 483
5/1/19

4. Error Analysis

Perform an error analysis of your best system. How many questions were answered correctly/incorrectly? Why do you think the correct questions can be answered by such a simple system? What problems do you observe for the questions answered incorrectly? Try to group the errors into a few classes and discuss them. Lastly, what is the impact of stemming and lemmatization on your system? That is, what is your best configuration: (a) no stemming or lemmatization; (b) stemming; (c) lemmatization? Why?

How many questions were answered correctly/incorrectly?

In my best system, 21 questions were answered correctly (79 incorrectly). Of the wrong results, 23 of them were in the top 10 documents offered by my system.

Why do you think the correct questions can be answered by such a simple system?

Looking through the 21 correct questions, some of the correct documents were Cairo, Duce (Itallian leader), Souvlaki, Knights of Columbus, Aaron Burr, and Ouzo. Something all of these have in common is that their document content is focused on categories surrounding the key words found in the query. There are page headings, followed by paragraphs containing the query words repeated, raising the document score for these. Specifically, Duce's score came out way ahead of the next highest document (A score of 40ish versus 32 for the second document), and on Duce's Wikipedia page, the query term Benito Mussolini is both a section heading and a link, in addition to being used in the paragraph, and "leader" being in quotes kept it from being shortened both in the lemmetazation and in the document parsing, so it found the "leader" on the Wikipedia page.

One of my changes was to involve the category, as described in Question 1, as just a word in the query. The query "Daniel Hertzberg & James B. Stewart of this paper shared a 1988 Pulitzer for their stories about insider trading" was wrong both without using category, and when using it as a separate field, first because it gave me something not newspaper related, and the second because it gave me a different newspaper. When I let newspaper be just a term in the query, I think that got me away from the simple "paper" term far enough to find the Jeopardy Question "What is... The Wall Street Journal?" Five other queries were found as well by involving the category in this fashion.

What problems do you observe for the questions answered incorrectly?

I'll break this up into 3 categories: Questions which did have the correct answer in the top 10, questions that didn't, and general remarks about limitations of Wikipedia results together with a couple of ideas how to mitigate them.

Questions which did have the correct answer in the top 10: Take for example the question whose answer was Three's Company. The query named Don Knotts contained details found in the summary of his acting history on his page. Together with Don Knotts repeated so many times and the summary of what happened in Three's company, it's no wonder that WatsonMinch guessed Don Knotts as the answer. Another few instances of guessing a term in the query were: Lord Byron, for which my program guessed John Keats since Keats was in the query, and George Martin, for which my program guessed the page "Fifth Beatle", which is all about George Martin, and "Fifth Beatle" was the clue given to find that page. The content from these queries was repeated in multiple pages, so the query obvioulsy chose the one that repeated the main query words the most, which would be the page with the name in the query.

Questions which did not have the correct answer in the top 10: Many of the questions that did not appear in the top 10 were clues that wanted the name of something related to an event, and WatsonMinch would instead give the event that it related to. For example, the category CAPITAL CITY CHURCHES gave a description of the church Matthias Church, and WatsonMinch gave one similar instead of a city. The query requesting a national cemetary instead gave the name of a Memorial commemerating something similar.

Now is as good as time as any to list the incorrect response that gave me the biggest laugh:

```
And the answer is:
> News flash!  This less-than-yappy pappy is sixth veep to be nation's top dog after
chief takes deep sleep!  1920s NEWS FLASH!
What is...  Kid Flash?
That's wrong.  Possible responses:
Calvin Coolidge
```

Hehehe... Poor Calvin Coolidge getting confused with Kid Flash. Anyways, back to the analysis:

General errors based on Wikipedia structure: There are two recurring errors that I saw popping up in the documents. The first was the huge amount of *List of ...* pages. These pages contain links to a whole lot of other pages, but are never an answer in and of themselves. For example for the clue, 11In the 400s B.C. this Chinese philosopher went into exile for 12 years HISTORICAL HODGEPODGE", WatsonMinch guessed: "What is... List of British Columbia provincial highways?" That was one of several that got the List page as a top result, and several others had List pages polluting the top results. Seeing this issue, I tried what I thought would be an easy query manually, and asked WatsonMinch the following original query: "This country

is home to 50 states, including Arizona, California, Florida, and New York." I thought this would easily return the United States, but instead it returned 9 documents that were all "List of United States..." and some topic, like Sports Teams, or Toll Roads. A way to circumvent this would have been when I indexed the documents to either remove all of the "List of..." pages, or possibly integrate them as additional document text to the pages they list.

The other recurring error I saw had to do with redirect pages. Since the "text" of a redirect page is only about 10 words long, it got scored really high if those words happened to be words in the query. Thinking about the application of my system, it would have made more sense to do one of two things: 1) Capture redirect pages, and link their text to that of the page they redirect to. 2) Just not count redirect pages. Either of those options would have removed several false flags, so hopefully that would get us closer to the real results.

The last problem I observed was a fault of my own system that I should have considered. I shouldn't have allowed the score to consider documents that contain the words of the query. Alex was never going to say "The answer is: This is the name of Joe Biden" and expect the question "Joe Biden". However, several wrong results that were given had terms in the query in common with the question generated. WatsonMinch could be improved by skipping documents for scoring that have a Document ID containing, say, more than 2 terms of the query.

Lastly, what is the impact of stemming and lemmatization on your system? That is, what is your best configuration: (a) no stemming or lemmatization; (b) stemming; (c) lemmatization?

I discussed my choice and how my program can handle optionally stemming and lemmatization, both for the queries and the Lucene Index. The most obvious result was in any of my setups, I always got the best results when my query cleanup matched that of the Lucene Index; that is, if I stemmed and lemmatized my Wikipedia documents before indexing them, then queries that weren't stemmed or lemmatized were largely unsuccessful (just 9 / 100 without category, 12/100 with), and if I left my Wikipedia documents unlemmatized and unstemmed before indexing, trying to query lemmatized queries on them also failed miserably (I accidentally deleted that Lucene index when making the stemmed one and forgetting to change my directory, but that one gave like 2/100 if I recall correctly).

With each addition of a feature (stemming or lemmatization, as well as adding stop word removal), my retrieval improved to a limit of about 15. It wasn't until I implemented the change on how I handle the category (described above) that I was able to get above 20 documents, which came after I started using the StanfordCoreNLP Stemmer instead. I swapped stemmers because something I was noticing with my TextLemmitizer class, is that the lemmenizedQueries it would add didn't get very well stemmed by Lucene. The "s" would get lopped off words that still had them after lemmatization, however, not much else happened. So I looked to see if the StanfordCoreNLP had any Stemming tools. It did, but not in the basic package. The instructions were to simply download the file from Github and include it in the classes, as there wasn't a standardized way to get them via maven. As such, I downloaded Stemmer.java from the StanfordCoreNLP github page, removed a few methods that were codependent with other files such as the Main method, and once I did that, I made a Stemmer object in my Text Lemmenizer class, and used the stemmed versions to put into the Lucene text body part of the index. This improved my performance by a couple of documents, so I kept it. That brought me up to my running best 21/100.

All in all, I still have 3 ideas that I've mentioned here about how to improve the queries some more, but it's still too simple of a system. I would need to go all out and categorize Wikipedia documents before hand and add rules for how to treat certain kinds of questions (Like returing a city when given a church in that city, etc). However I am very happy with how the program turned out.