Ruben Tequida
CSC483

Final Project – Building (a part of) Watson

Instructions:

- Clone the repo from github
- Download the preconstructed indexes from this dropbox folder and put each individual folder in the resources folder (src/resources/)
    - If you want to build the indexes yourself you will need to download the repo of Wikipedia pages from here
- From here you can either utilize the .jar (located in: out/artifacts/Final_Project_jar) to run the program or run Main.java directly
- The program will prompt you to choose how you want to handle the content therefore choosing the proper index and how the queries are handled (lemmatization, stemming, or neither)
- Then, it will prompt you to choose which scoring method you want to use (default (BM25), Boolean, Jelinek-Mercer, or TF-IDF)
- It will then display the percentage of answers guessed correctly using precision at 1 and then display the mean reciprocal rank
- Lastly if will ask you if you want to repeat the process

Indexing and Retrieval:

Index Creation: The index is created within the IndexGenerator class. This class makes/opens each directory depending on which style of processing you selected. Then, it goes through each file in the Complete_Data_Set folder and separates each Wikipedia page by finding the title section of each page. Once a new title is found it will put the title in the index as a StringField so no tokenizing is done to it and it will add the page categories and it's content to the index as a TextField so these are tokenized. Before adding to the index, the IndexGenerator will remove TPL tags that are found within the content of the Wikipedia page because those offer no useful information pertaining to answering the questions.

What issues specific to Wikipedia content did you discover, and how did you address them?
I found that Wikipedia pages have a good amount of formatting that does not apply to indexing the page such as tpl tags and sections, categories, and additional sections in the page. I address the tpl tags and sections by removing the tags and the sections between them from the line being added to the index. I handled the categories section by removing the header of category since it is irrelevant to any queries and adding everything after it in that line to the index. Finally, I address the additional section headers by specifically looking for them and skipping over them if I found them.

Query Handling: The questions file is opened, and each question is handled individual within the QueryDissector class. This class separates the question into category, clue, and answer. Then, performs processing on the category and clue so the query is processed the same way the content in the index was. Next, the QueryDissector queries the index using the query created from the category and clue and determines if the guess with the highest score was the correct answer and if it wasn't determines if the correct answer is within the top ten guesses returned by the INeedAnswers method.

Describe how you built the query from the clue:
I built the query by adding the category to the beginning of the clue and then stylizing the tokens based on how the user chose to stylize the tokens that were added to the index (i.e. lemmatized or stemmed) and then set that entire string to lowercase.

Are you using the category of the question?
Yes, I used the category of the question by adding it to the beginning of the clue and then making the query from that. I figured the category would help improve scores since I added the categories from the Wikipedia pages to the index as well.

Measuring Performance:

Justify your choice, and then report performance using the metric of your choice:
I chose to use precision at 1 (P@1) and mean reciprocal ranking (mrr) as means to measure the overall performance of my indexer and query parser. I used precision at 1 because that would give me a total percentage of questions that I got right out of 100 and quantifies the programs performance in a way practically everyone is used to seeing and comprehending percentages. I then chose to use mean reciprocal ranking because it allowed for some flexibility in choosing the correct answer instead of simply always going with top ranked answer. As you can see with Table 3.1 using the mean reciprocal ranking gave the program more leeway by allowing it to search the top ten guesses for the correct answer.

Changing the Scoring Function:

I implemented four different ways to score the queries and allowed the user to choose which method they wanted to use. The methods I chose were default (BM25), Boolean, Jelinek-Mercer, and TF-IDF. The results after going through all 100 questions were as follows:

Table 3.1

Scoring \ Style	Lemmatization		Stemming		None	
	P@1	MRR	P@1	MRR	P@1	MRR
Default (BM25)	20.00	26.16	22.00	28.26	17.00	23.34
Boolean	10.00	14.10	14.00	16.70	8.00	13.17
Jelinek-Mercer	25.00	31.51	24.00	30.27	26.00	31.15
TF-IDF	1.00	2.94	2.00	3.88	2.00	3.35

How does this (the scoring function) change impact the performance of your system?
As you can see from Table 3.1, changing the scoring function can drastically change the performance of the system. Jelinek-Mercer smoothing with mean reciprocal ranking worked the best in providing a correct answer. The default scoring method of BM25 did not do as well as the Jelinek-Mercer method, but still did rather well compared to Boolean and TF-IDF. I expected Boolean to perform rather poorly since it is very simple, but I did not expect the result I got from the TF-IDF method. This could have been caused by the fact that I used the similarity of ClassicSimilarity, which is a subclass of TFIDFSimilarity, and could have cause inaccuracies when performing queries.

Error Analysis:

When using Jelinek-Mercer and no lemmatization or stemming I got the highest percent correct with precision at 1 with a total of 26 out of 100 questions answered correctly. However, when using mean reciprocal ranking with Jelinek-Mercer and lemmatization I was able to increase the chances of the system guessing the correct answer 31.5% if it were able to search the top ten scoring answers returned when querying the index.

Why do you think the correct questions can be answered by such a simple system?
I think this is possible because the questions had clues that are based on the Wikipedia pages the answer comes from and therefore can be very similar in structure and word use to the content that was added to the index when they were built. With the inclusion of categories and since the answers are Wikipedia page titles this system has a chance of answering the question correctly because it can narrow down the possibilities based on the key words given in the categories section of the Wikipedia page along with the categories given inside the question. It can also be noted that the system we are using is not very complex as we are determining an answer simply by analyzing the similarity between the clue given and the content of the Wikipedia page. Using a simply system that has been modified over time to increase performance while not increasing complexity has a high chance of performing well at specific takes and also minimizing errors and keeping out irrelevant data and information.

What problems do you observe for the questions answered incorrectly?
By grouping the errors into two classes (clues having similar topics therefore having very similar verbiage and clues that are short) there can be a better understand of why errors may occur. If a question or clue utilizes words that span multiple Wikipedia pages or their general topic falls under the umbrella of many of these pages as well then, the ranking system may not be very effective in producing the correct answer. When the clue uses a lot of common words or even specific words that strongly relate to specific categories then the indexer could produce many results that are very similar to the query but are not actually the answer to the question. The same thing occurs when you have a lot of Wikipedia pages that are about the same general topic so when a query comes in that uses common words for that topic it may be difficult for the index to accurately determine which page you were actually looking for. Another issue that can occur is when the clue given to an answer is too short. If the query is too short then there aren't many data points to compare against the index, therefore, leaving much more room for error the short query may be composed of stop words or very common words or even a few words that appear in many different Wikipedia pages so hardly any narrowing down of pages occurs.

What is the impact of stemming and lemmatization on your system?
As seen in Table 3.1stemming typically performed better if not equally to lemmatization, however using neither stemming nor lemmatization produced better results when using MRR and Jelinek-Mercer smoothing. For best results the recommended settings would be either lemmatization with MRR and Jelinek-Mercer, stemming with MRR and BM25, or neither stemming or lemmatization and P@1 and Jelinek-Mercer.