Adrian Bao
CSC 483 - IBM Watson
12/09/2020

0 Instructions

Detailed instructions (including commands to run) can be found on the Github README found here: https://github.com/BaoAdrian/IBM-Watson

However, here is a brief synopsis to get up and running

1. Clone the repo: https://github.com/BaoAdrian/IBM-Watson
2. (Optional) If you wish to use pre-built indexes, download the zipped manifest here: https://drive.google.com/file/d/1CABlrkPPYIvkIZ6HzgijiUYaIRbz9bQ1/view?usp=sharing
    a. Place zip into the IBM-Watson directory and unzip it
    b. ** See Github README for expected file structure at this stage **
3. If you chose to skip (2) OR wish to test *IndexEngine*'s ability to generate a Lucene Index, you will need to download the wiki data tarball from here: https://www.dropbox.com/s/nzlb96ejt3lhd7g/wiki-subset-20140602.tar.gz?dl=0
    a. Once downloaded, move tar ball into `src/resources` directory and untar the tarball (`tar -zxvf wiki-subset-20140602.tar.gz`)
    b. ** See Github README for expected file structure at this stage **
4. Open the project in your favorite editor and compile the project, run the driver, `IBMWatson.java`
5. Proceed with running the program. It will start by asking you which method you'd like to build the Index then it asks you which method you'd like to use for queries. Once the program completes processing the files and performing the query, it will calculate the P@1 score and display it on the console. Finally, it will prompt if you wish to continue again to process more index-building or querying tests.

1 Indexing & Retrieval (100 pts)

1.1 Overview

The objective of this project was to replicate IBM Watson's leveraging Lucene & CoreNLP to provide an Engine that builds a Lucene Index based on Wiki Data (provided) as well as providing the ability to *query* the Lucene Index using various similarity algorithms. Additionally, we wished to analyze whether or not pre-processing the wiki data when generating the index has any influence on the overall performance of IBM Watson, i.e, whether Lemmatizing, Stemming, or leaving the text, as-is, achieves lower or higher probabilities at finding the correct answer.

Two major components make up this project:

- *IndexEngine*: Builds the Lucene Index

    - Supports: Lemmatization, Stemming, & Neither Lemmatization/Stemming (raw)
- *QueryEngine*: Queries the Lucene Index built/provided by the *IndexEngine*
    - Supports: *BM25, Boolean, TF-IDF, Jelinek Mercer*

1.2 Indexing

1.2.1 Describe how you prepared the terms for indexing

The first portion of this project required us to create an *IndexEngine* to build a Lucene Index using various methods such as *Lemmatization & Stemming* (or neither). Generating the Index includes adding Lucene Document objects in the following format:

```
Document {
    StringField: title
    TextField:   category
    TextField:   textAttr
}
```

- `textAttr` above represents an indexable field that concatenates the *category*, *header*, & *text* values parsed from the wiki data to add possible matches when querying

The process of reading from the wiki documents & generating the Lucene Index is done in two main steps:

1. Extracting key attributes from Wiki documents, namely *title, category, header & text*
    a. This was a bit of an intricate process as we need 'clean' or 'sanitize'' content to enhance the *QueryEngine's* performance. This included having to remove much of the extra characters that exist and/or surround the key text we are looking for.
        i. Ex: `'[[Some Title]]'` needed to be `'Some Title'`
    b. This also included removing `tpl` tags as these occurred throughout the wiki documents and provided no real value to querying
2. Processing the extracted attributes using the user-selected method (Lemma, Stemming, or Neither)
    a. CoreNLP was very helpful here since they provide a `Sentence` object that does some tokenization processing for us and provides us with access to `lemmas()` and `words()`
    b. Additionally, when the user selects *stemming*, a `PorterStemmer` is utilized to iterate over just the stems of the words in the Sentence

You will see that the pre-processing of the wiki data using these methods does actually influence the overall performance of IBM Watson as some increase the likelihood of a correct answer being retrieved by the *QueryEngine* (see INSERT section).

A brief synopsis of how each method's preprocessing is done within the *IndexEngine* is as follows:

Lemmatization

- `StringBuilder` is used to cleanly aggregate Strings for our key attributes

- CoreNLP `Sentence` object is used to read in key attributes and allows us to iterate over the *lemmas* using `sentence.lemmas()` and appending them to our `StringBuilder`
- Aggregated Strings/key attributes are then added to a Lucene `Document` object and written to the Index

Stemming

- `StringBuilder` is used to cleanly aggregate Strings for our key attributes
- A `PorterStemmer` is used to extract the *stem* of each word that we process
- CoreNLP `Sentence` object is used to read in key attributes and allows us to iterate over the *stems* using `sentence.words()` , using the stemmer to extract the stems, and appending them to our `StringBuilder`
- Aggregated Strings/key attributes are then added to a Lucene `Document` object and written to the Index

Neither Lemmatization nor Stemming

- Raw strings for key attributes are kept as-is, added to a Lucene `Document` object and written to the Index

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

Fortunately, the Wiki documents that were provided were in a relatively predictable format allowing us to have rudimentary extraction of key attributes we were looking for. There is, however, plenty of additional information that can pollute the performance of IBM Watson, namely tpl tags, ref tags, hyperlinks and other things of that nature that do not provide any positive value to querying.

1.3 Retrieval

1.3.1 Implementing Querying (query as clue, title as answer). Describe how you built the query from the clue?

While building *QueryEngine*, my approach was to process the clue parallel to how the text was processed by *IndexEngine*. I opted to not overcomplicate the processing of the clue just to see what the initial results returned which were actually not too bad.

For starters, we were given the format of each question in the document as follows:

```
1 | Category
2 | Clue
3 | Answer
4 | newline (empty space)
```

Extraction of these items was simple as the format was strictly defined. One thing to note in here is that I chose to concatenate *Category & Clue* to be the actual *query* that we run against the Lucene Index. I opted to do this simply because the *Category* (when combined with *Clue)*

can provide some positive value when performing the query and may lead to higher hit-rates than just using the *Clue* itself.

Just like with *IndexEngine*, depending on what the user selects as their index method (Lemmatization, Stemming, or Neither), I chose to process the clue in the same exact way (see synopsis above for the overview of how the Strings are processed).

Next, the *Category + Clue* are parsed using a `QueryParser` which is then provided to the `IndexSearcher` to search for the top 10 results. I chose 10 results as this seemed to be what we used in homework 3, plus the other results weren't used in the metrics I chose to analyze.

2 Measuring Performance (25 pts)

The metric I chose to use to measure the performance of this Engine was **P@1**, or Precision at 1, meaning which calculates the percentage of all questions in which the Engine was able to retrieve the correct answer as the top result. This was accomplished by having two basic counters, one counting the total number of questions processed, and the other counting how many times the top result matching the expected answer.

This metric seemed to provide the raw, face-value performance of each method combination supported by the program. The overall objective was to see, using various indexing/querying methods, which combination(s) yield the *highest number of correct answers*. In real jeopardy, only the top answer is considered, therefore, **P@1** seemed like a logical metric to use to measure how many times the system correctly retrieves the correct answer as its *top* result.

Metrics for all methods can be found in the following section...

3 Changing the Scoring Function (25 pts)

As mentioned before, a total of *four* similarities are used:

1. BM25
2. Boolean
3. TF-IDF
4. Jelinek Mercer (using a lambda of 0.5 - often used in class)

Each model was run against indexes built using Lemmatization, Stemming, and *neither* Lemmatization nor Stemming. This yielded the following performance metrics when calculated the **P@1** scores for each:

%		Query Method (Similarity Algorithm)			
		BM25	**Boolean**	**TF-IDF**	**Jelinek Mercer**
Index Method	**Neither**	0.21	0.1	0.0	0.28

	Lemmatization	0.19	0.13	0.0	0.28
	Stemming	0.21	0.1	0.02	0.26

A few interesting conclusions can be drawn from the above metrics:

- The *Jelinek Mercer* similarity model was, by far, the best model used in these tests
- The *TF-IDF* similarity model was, without question, the worst one used, which was expected
- It appears that BM25 performs *better* when the index is generated using either *Stemming* or nothing at all. Conversely, BM25 performs *worse* if the index is built using *Lemmatization.*
- It appears that the Boolean Similarity performs *better* when the index is generated using *Lemmatization* but performs *worse* when the index is built using *Stemming* or nothing at all.

4 Error Analysis (50 pts)

4.1 How many questions were answered correctly/incorrectly?

Referencing the table above, those scores are generated out of the 100 total questions processes, so for example, BM25 using lemmatization found the correct answer 19 of 100 times.

If we specifically look at our best score(s), Jelinek Mercer using either lemmas or nothing at all yielded the highest number of correct matches at 28 out of 100 questions.

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

It appears to be a simple system, however, the 'heavy lifting' provided by the immensely complex libraries like Lucene & CoreNLP help alleviate the development burden. It allows us to simply 'sanitize' the input that these libraries expect and they handle the rest. What also helps increase the number of correct answers being found is the various avenues of information provided by Wikipedia, that being the Categories, Headers, and Text. Combining all of these *unique and separate* fields into one, indexable, field for the index allows for better query performance overall, and this is seen in the table above.

4.3 What problems do you observe for the questions answered incorrectly?

With the questions answered incorrectly, an influence I definitely think is negatively affecting those results are stop-words. As implemented, there isn't anything set in place to remove stop-words, which therefore, provide positive scores for matches against those irrelevant words to non-related documents. This influences what is ultimately chosen as the top answer and when both the correct answer's *clue* and the incorrect answer's *indexable data* has lots of

matching stop-words, it bumps up that incorrect answer, ultimately selecting it as the top answer.

If the *QueryEngine* further processed the query to remove stop-words, then the engine would be able to isolate *only* the relevant information included in the clue and, therefore, have a higher chance at selecting the correct answer as its top search result.

4.4 What is the impact of stemming and lemmatization on your system? Best settings?

Interestingly enough it appears that the highest statistics for P@1 are when *Jelinek Mercer* is used without Stemming/Lemmatization, just by a small fraction. In general, in larger sample sizes, stemming and lemmatization ideally would improve performance given they help in searching the index. Lemmatization *did* improve the performance in Boolean, however, have a more positive impact in that query method.

Overall, the most performant run of the system is tied between *Jelinek Mercer* on an index built *without* either Stemming or Lemmatization OR on an index built *with* Lemmatization, both of which resulted in 28 correct matches out of 100 total questions.