CSC 583 Final Project

Sarah Stueve

Dr. Mihai Surdeanu

Fall 2020

# Part 1: Indexing and Retrieval

## How to run

Here is a link to the GitHub repo for my project: <insert link here>. This code is formatted as a Maven project and should be loaded as such. Required dependencies have been added to the pom.xml file and should be able to load on creation.

You will need to download my best index from this box link and place it in the resources directory/package:

<https://arizona.box.com/s/v8s0pi0ulzq2y137oqpf8yyg5b1utm1b>

Everything was done using Lucene packages.

## Description of the code (includes info re: indexing and retrieval)

The source code for this project is separated into five distinct classes: index, babyWatson, driver, Question, and ResultClassWatson (sorry for the inconsistency in class name format – Java is new to me!).

Question and ResultClassWatson are used as containers – Question instances contain information about the jeopardy questions to be answered, including: a generated question ID, question text (which is parsed to include the category), and the question answer. Similarly, ResultClassWatson instances are used to house prediction information and contain variables for: relevant question id for answered question, predicted document, and document score.

The index class is used to build the index, using the buildIndex method. The index is built in the type FSDirectory and each document in the index contains the document (wiki) text and the wiki name. In order to build the index, the data was parsed by reading each of the individual 80 files containing wikis into strings and then parsing them by splitting the strings largely on newlines – I found this was the easiest solution, as the lengths of the documents were variable. I was able to do this using the solution from [this StackOverflow entry](https://stackoverflow.com/questions/15161553/how-to-convert-fileinputstream-into-string-in-java). I decided to include the document categories in the text but made sure to remove the word “categories” so it was not used as a predictor, though since it would have appeared in most of the documents, I doubt it would have made much of a difference. The initial analyzer I used for the creation of the index and question answering was the StandardAnalyzer.

The class babyWatson is where the questions were processed (method: questionParser) and predictions (questionAnswerer) were made. The questions were stored in an ArrayList of Question objects, one for each question. In order to preprocess them so they would largely match the documents in the index, the questions were stripped of any non-alphanumeric characters apart from the @ character (I did not preserve select periods) and split on spaces. The category of the question was added to the beginning of the question list for that question and, initially all tokens were kept. While initially I chose not to run a feature selection algorithm in the query because I thought it made sense to first try to make predictions on the entirety of the query, without adding ambiguity to tokens selected, I decided to make that change as part of the improving retrieval portion of my project. [See that later in this PDF](#_Part_5:_Improving).

The questionAnswerer method takes as input the list of Question objects and the document index and parses the queries using Lucene. The return value ans is a list of lists containing ResultClassWatson instances. Each inner-list contains the top 10 documents retrieved for the question.

driver contains the main program. First, it either loads or builds the index, depending on whether the index already exists. After the index is loaded, a babyWatson instance is created to get the predictions for the questions. The gold data are also added from the babyWatson parsedQs instance variable, which contains the gold answers for each of the questions. From there, the gold and predicted answers are compared and the Precision @ 1 score is calculated.

# Part 2: Measuring Performance

A Precision @ 1 score was chosen because in this case we don’t really care about the order of documents returned by the system (which are ignored by this measure, but we only care about the one document that gives us the answer to the question) and it’s easy to calculate manually. The baseline retrieval system retrieves relevant documents (out of the documents retrieved) approximately 16% (precision = 0.16) of the time. The retrieval system using stemmed data, which was the best system when compared to lemmatization and a combination of the two, retrieves relevant documents approximately 17% of the time (see below)

# Part 3: Changing the score function

Replacing the default BM25 scoring Lucene uses with tf-idf scoring drastically decreased the precision of my system, by approximately 0.15.

Precision (BM25): 0.160000

Precision (tf-idf): 0.010000

# Part 4: Error analysis

The error analysis for this portion will be completed for the default BM25 ranking system.

There are a few categories that immediately stand out in the error analysis. Using the BM25 retrieval system, there were 16 correct answers and 84 incorrect answers. One very common category of error I can see is names/wiki articles that have multiple potential spellings or names? I did not handle those cases in preprocessing and handling them may improve performance. One potential way I could see doing this might be by choosing the first one or maybe selecting one using feature selection? But I’m not sure which is more feasible. Another common theme observed was the system retrieving documents about a related person or thing to what the category was referring to, but not the correct version. For example, Michael Jackson was selected multiple times for 80’s related retrieval questions. There were many that fall within this category of retrieval error and I can see how it may be difficult to differentiate those documents, especially if each of those similar documents contain references to each other. There were also many, related to the previous category, where the same country was mentioned in both of the document names but they were about two different topics – one way to combat this may be to bias the system against country names, only slightly, to make sure the system also weights other words.

Below are the results from a series of tests on nothing, stem, lemmatized and stemmed and lemmatized data:

Precision (nothing): 0.160000

Precision (stemmed): 0.170000

Precision (lemmatized): 0.150000

Precision (stemmed and lemmatized): 0.170000

The best index was tied between the stemmed and stemmed and lemmatized indices gaining a single percentage point in precision. The stemmed index was chosen as the best index because it generated the better system performance with less preprocessing steps. It’s possible stemming might improve performance over lemmatization because stemmed words may be more likely to match several tokens, given the stems of multiple different words might be the same, versus lemmatization which always generates real words and isn’t quite as strict in terms of preprocessing. Lemmatization appeared to have no affect given previously stemmed words, but that may be because I lemmatized after I stemmed rather than the other way around.

# Part 5: Improving retrieval

In order to improve retrieval, I attempted to create a feature selection algorithm that selected the top 5 most common words in the question in the relevant wiki’s text however was unable to complete it. The code that was partially created can be found in the FeatureSelector class.