Kyle Raidl
Project 1 - Watson
karaidl@email.arizona.edu

To run the code, simply execute the program and follow the command line prompts.

Please note, lemmatization is extremely slow. I do not know why, although it must be because of my algorithm. That being said, lemmatization does not seem to offer a benefit. More on this later in the paper.

Problem 1.
My system has three basic configurations when indexing the documents. The configuration names are “Good”, “Better”, and “Best”. Good uses the Standard Analyzer, which lowercases each token and removes common words and punctuations. Better takes categories into account by adding them to the query. Both Good and Better are also capable of using the Whitespace Analyzer, which simply indexes all characters between whitespace, and does not remove anything or change case. Best takes categories into account and uses the English Analyzer which takes the Standard Analyzer and provides stemming procedures as well. I retrieved a list of stop words from an online posted by Yoast SEO, a search engine optimizer for Wordpress. Everytime a term is checked against a query this term is first checked against a HashSet of stop words. If it is a stop word, then the engine moves on to the next term in the document. Stop words are still indexed as not to mess with the scoring within the documents.

One of the first issues I encountered when attempting index the wiki data was that the section titles were often redundant to the rest of the text and essentially stop words, because they would be seen in every wiki article. For example, there is no good reason to index the “references” title of any article. Another issue I ran into was what to do with any external links. Because the Standard and English Analyzer both will strip out punctuation, I decided to go ahead and leave the urls in the collection. However this will most likely cause issues with the Whitespace Analyzer, as these urls are very unlikely to be relevant when punctuation is included.

In order to turn a clue into a query, the entire clue is included in the query. Stop words are eliminated, which means only a subset of the query is utilized. If using the Best configuration, then categories are also included in the query. Beyond this, no algorithm has been developed to further eliminate terms from the query.

Problem 2.
To measure the performance of the system, I decided to estimate the precision. To do this I took the first question, which corresponds to The Washington Post, and I used it as a means of measuring precision. Overall, I managed to find 267,680 articles which were relevant to the query. That is, at least one keyword within the query (not including stop words) was found within a relevant document. Next I returned the number of hits I received from my Best configuration

with no lemmatization, which was of size 99,986. I kept both of these hits within two HashSets and compared them against each other and found the intersection, then took the size of the intersection, which was of 92,287. Computing this, I found that this configuration will produce a precision of 92.3%.

Problem 3.
I replaced the default scoring function within Lucene with the probabilistic model of BM25. Furthermore, after some experimentation I have determined a good adjustment of its similarity profile. The similarity is the scoring function for which BM25 is based upon. To adjust it, two Floats are needed - one which specifies non-linear term frequency normalization, and the other which controls to what degree document length normalized term-frequency values. BM25 retrieves a lower degree of document, but with a higher degree of precision. 75,716 documents were retrieved, of which 73,293 were considered relevant. Therefore this model has a precision of 96.8%.

Problem 4.
The absolute best system I have to offer is the Best configuration along with BM25 scoring, with adjusted weights! Using this configuration, the error rate is 63% with 37 questions answered correctly.

Interestingly, articles which corresponded to a person's name would often be the most accurate, but there were many names which did not correspond to a correct answer. Examples of these names include Michael Jackson, Heath Ledger, and Irving Berlin. In the case of Irving, for example, the returned result was "My Funny Valentine", a movie he starred in. No doubt this is why this article was retrieved. All articles which were retrieved incorrectly but were supposed to return a name, almost always retrieved a topic which was relevant to the person.

However, when a result is returned correctly, I believe it is because the most relevant information in the query will be a frequent term within the document.

Another type class error was whenever the query corresponded to a location, oftentimes the retrieval system would pull a date, event, or person which is highly relevant to that location. Examples of this include Helsinki, which returned John of Cappadocia. There was also France, which returned the Politics of Cambodia, which makes sense given the French involvement in Cambodia over the years.

The best system in my program is to use stemming, but no lemmatization. For one reason, this is because lemmatization takes a brutally long time - over three hours on my computer! I am not certain of why this is, although it is obvious I must not be implement my lemmatization algorithm very efficiently. I have attempted to refactor my code to use the SimpleCoreNLP library instead, but have found this too cumbersome when close to the deadline. Furthermore, lemmatization does not actually offer any increased accuracy. I believe this is due to the simplicity of the system. When only 37% of results are accurate, it is unlikely that returning terms to their base

form will boost accuracy. However, stemming most certainly does make a difference. Without stemming, only 29 results are returned correctly, versus the 37 correct results with stemming. This probably makes a difference, because again, this system is very simple and stemming is a simple, if not crude way of reducing a term to its base form.

Here's a quick rundown of functions provided within the program -

CORE NLP LEMMATIZARION FUNCTION. ACCEPTS STRING -> ARRAYLIST OF LEMMATIZED WORDS
public ArrayList<String> lemmatize(String documentText)

FUNCTION TO CHECK IF THE TOPHIT DOC HAS A WORD THAT IS PRESENT IN THE QUERY/QUESTION
public boolean isPresent(String topHit, String question)

FUNCTION CALLED WHEN INDEX OPERATION IS CHOSEN
private void indexData(String indexPath)

FUNCTION TO GET ALL DOCUMENTS PRESENT IN THE FOLDER CONTAINING THE WIKI DOCUMENTS AND INDEX THEM ONE BY ONE
private void indexDocs(IndexWriter writer, Path docDir)

FUNCTION THAT PARSES THE CONTENT IN EACH DOC AND SENDS TITLE AND CONTENTS FOR INDEXING
private void indexFile(IndexWriter writer, File file)

FUNCTION TO ADD TITLE AND CONTENTS
private void addDoc(IndexWriter w, String title, String content)

FUNCTION TO SEARCH INDEXED DOCS BASED ON THE QUESTIONS
private void searchIndex(String indexPath)

FUNCTION TO GET THE TOPHIT DOCUMENT FOR THE GIVEN QUERY
public String getTopHit(IndexSearcher searcher, Query query, String quest)