IBM’s Watson Project Report

1 Code description

The application we created is a console application made in Kotlin and with the help of Lucene, where users can input a number corresponding to a command from the displayed menu.

In this context, there are three possibilities: 1. creating the index; 2. providing a custom query to search using the created index, or 3. running the default questions of the project.

Now, let's review the createIndex function. Initially, we need to delete the file of the previous index (if it exists, of course). To achieve this, we call the deleteFilesFromDirectory method that we have implemented. This method essentially deletes all files from the specified path parameter. After this step, we obtain the analyzer. Since we require the same analyzer for both indexing the documents and retrieval, we have a function that returns the analyzer. This function is utilized throughout the whole code.

After configuring the index, we proceed to parse through the collection of Wikipedia pages. This is done by each file in processFile function, which have the following implementation: we initialize an object of type WikiPage, then we start to read line by line, being interested only in not blank lines. Firstly, we check if the line is a title line, which means it has the following format: [[<Wikipedia page title]]. If the title that we previously configured is alright and the page was not a redirect to another, then we are in the case when we finished to parse a page and have to added it to the set of all pages (there is also implemented a comparator for two WikiPage, they being compared by their titles) after setting the property for body. This can be seen as a stop condition when we reach a new Wikipedia page in the file. After this, we have to make the configurations for the new Wikipedia page, by setting the current title that we found and an empty body.

Otherwise, if the current line is the one for categories, we will parse it and get all categories as a list of strings. If the current line is a redirect line (meaning it starts either with #REDIRECT or #redirect), then we reach a Wikipedia page that has no body but only redirects to another page. We will keep this information in a map where the key is the redirect link and the value is a list with all pages that redirected to that page.

If we are not in one of these conditions, then we just have to append a new line to the body of the current Wikipedia page. After we finish to parse the file, we have also to add the last Wikipedia page in collection. Because in Kotlin, the parameters are passed by value, we have at the end of processFile function to do some adjustments for setting the collections with all data that we have until now.

Going back to createIndex function, after we finished to parse all the files and have a collection with all the Wikipedia pages (that were not just redirect pages to another) and a collection to handle the redirect pages, we can create the index.

For each Wikipedia page (that is not a redirect to something else), we will create a document in the index that will have 3 fields: 2 of them being StringField (title and category, their content won't be tokenized) and 1 one of them TextField (body, its content will be tokenized). Both for title and category we can store multiple values, and here comes the part where we use the collection of redirection pages. If there are more pages that redirect to one, that one page will have multiple values for title. After we created the document for each page and added to index, we saved the index to its path.

The runUserQuery takes from the console the text that the user wants to search in the Wikipedia pages, then constructs the query by only using the "content" field of the index that we created and there are looking for only the first 10 results that will be printed. For each document, we will retrieve a list of titles where on the first position is the title of Wikipedia page on which we created that document and the next ones are from the redirect pages.

The runDefaultQuestions parse the document with the 100 questions, and for each question there is constructed the query. Firstly, we take the category, clue and answer, then there are removed the special characters from the category and clue, as there were errors when they were present, and after this there is called the

method runSingleQuery. In this method, we create the query for each question, by using now both the “content” and “category” fields and the query will be something like “<clue> or <category>”. In the same time, we want to keep the category simple, so we remove some additional content in parenthesis (like for STATE OF THE ART MUSEUM). Then, we will look for the first 30 results from the index, print each result, together with the clue and the correct answer and the function runSingleQuery will return an Int that represents the rank of the correct answer in the retrieved documents. In order to see this, we take the list of titles for a document and make an intersection with the list of answers (for example, if the answer is JFK|John F. Kennedy, the list of answers will be the one with JFK and John F. Kennedy, being in the fact same pages with different titles).

Back to runDefaultQuestions method, we take all the ranks and compute the Mean Reciprocal Rank and also display the number of times when the first document was also the correct one. When computing the MRR, we considered that if a document has the rank 0 then it was not found in the 30 results, which means its reciprocal rank should have a small value, so we attributed it a value of 0 for these cases (it will be a mistake to remove from the whole computation these cases, as we would obtain better fake results)

2 Results

The output of our code can be seen in the file Results.txt within the Git project. The questions are grouped by specifying which were the clue, the category and the expected answer as input data and after, for each question, which was the query answer, the result that we provided. Also, there can be seen which was the MRR that we obtained and the total number of times where the answer provided by us was the right one, at the end of the file.

In other words, we obtained that the MRR is 0.37 and there were 30 number of times where the answer was correct.

3 Explanations

3.1 Indexing and retrieval

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

To prepare the terms for indexing, we utilize a specific analyzer from Lucene called EnglishAnalyzer. As implied by its name, this analyzer is designed specifically for the English language. Based on an extension of the StandardAnalyzer, the EnglishAnalyzer removes possessives (trailing 's) from words, incorporates the Porter Stemming algorithm for stemming, and excludes common English stop words.

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

One aspect that drew our attention was the category lines. Initially, we considered including them in the 'content' field of the index. However, upon further consideration, we recognized the potential usefulness of this information for retrieval. Subsequently, we parsed these lines and observed that there could be more than one category. Consequently, we decided to allow multiple values for this field.

Another challenge arose from pages that solely served as redirects to other content. Initially, we chose to disregard them entirely. However, upon reconsideration, we decided to enhance our approach. For each document, we modified the 'title' field to be represented by a multi-value field. The first value corresponds to the title of the original page, while subsequent values include the titles of pages that redirect to the original one.

Describe how you built the query from the clue.

To construct the query, we considered both the 'content' and 'category' fields from the index, and additionally included the category of the clue. Our query takes the form '<clue> OR <category>' and searches across both the 'content' and 'category' fields of the index. However, we encountered issues with certain categories, such as 'STATE OF THE ART MUSEUM' (Alex: We'll give you the museum. You give us the state.), where using parentheses didn't make sense. To address this, we decided to streamline the information by extracting only the substring until the first occurrence of '(' for the category. Furthermore, to simplify the clue, we removed special characters.

3.2 Measuring performance

To assess the performance of our system, we employed the Mean Reciprocal Rank (MRR). We found that this measure aligns well with our problem because we can retrieve more answers for a query with sorted results by their scores and also the input doesn't provide a predefined list of relevant answers for a query, but rather a single correct answer. In cases where there are synonymous titles like 'JFK|John F. Kennedy,' they essentially represent the same content, being in fact one page with different titles. Consequently, MRR is particularly suitable for our scenario as it seeks to identify the rank at which the correct answer occurs, and then make a mean with their inverse. This approach allows us to retrieve multiple answers and determine if the expected answer is among them, helping us evaluate its position in the ranking.

There is a potential issue here in cases where the correct document is not retrieved at all. To address this, we set the limit to 30 hits per query. Initially, we used 10 hits, but we realized that there are many other documents with a rank between 10 and 30. In instances where the correct document is not retrieved, we consider using the value 0 for the reciprocal rank (note: reciprocal rank, not the rank itself) in the formula. Why? Let's consider a scenario where the document would eventually be retrieved at rank 1000. Its reciprocal rank would be 1/1000, which is nearly 0. The larger the rank, the smaller the reciprocal rank. If the correct document is not retrieved within the first 30 results, we can assume it has a rank of infinity,

resulting in a reciprocal rank of 1/infinity, essentially equaling 0. This approach helps us address the challenge of documents not being retrieved. **In this, we obtained a MRR of 0.374**

Simultaneously, we utilized a statistic to track the number of times the first result was the correct one. In this analysis, we found that **in 30 out of 100 cases, the first result proved to be the correct one.**

3.3 Error analysis

There were 30 questions answered correctly and 70 incorrectly.

We think that such a simple system was able to respond in 30 cases correct because of the implementation of Lucene index, in which a very similar structure can be found in the correct document. This would make text-matching to see the documents in which the clue was found. In the same time, we have to take into account the EnglishAnalyzer, which made an awesome work by stemming the words and removing the stop-words.

For the questions answered incorrectly, we can make 3 categories:

1. Questions of which answers were partially good as the correct answer was in the first 30 results - in this case, the index and retrieval should be adjusted in order to increase the number of times where the first result was the actual answer, but in the same time, it wasn't like the document was not found at all
2. Questions in which the retrieved answers were related to the same subject but didn't contain at all the exactly right answer – like the cases for Michael Jackson questions in which there were a lot of pages related to music and 80s, but the content page of Michael Jackson didn't contain the exact structure like 1983: "Beat It"; maybe a problem was because of the small clue?
3. Questions in which the clues was formulated in a natural language form and the right answer didn't contain the clue or even a subset of it, like it's the case for the question: Song that says, "you make me smile with my heart;

your looks are laughable, unphotographable" with the answer “My Funny Valentine” – the Wikipedia Page didn’t event contain the specified lyrics