IR Assignment 1 - Indexer

Anirudha Desai

October 6, 2017

Description of the System

The system comprises majorly of the following parts :

1. Main : Contains the main class. Controls the flow of the whole program in sequence.
2. FileExtract : Component for extracting the json from the gzip. Also, reads the json and collects the details of the play.
3. ExtractIndexTerms : Continues the pipeline from fileExtract. Extracts and builds the inverted List for every term. It handles all the postings to the inverted lists.
4. DiskWriter : This component handles the writing the inverted List and all other maps to disk.
5. RetrievalAPI : This component furnishes the APIs for evaluation activities.
6. Evaluation : This component serializes the evaluation components asked for in the assignment.
7. DiskReader : This component reads all the files and maps written to disk by the DiskWriter class.

The following retrieval APIs are published :

1. getFrequenciesForTerm : This API takes as input a String and returns the term Frequencies and Document Frequencies for that term.
2. RetrieveQuery : This API takes as input a query string and the number of retrieval results, k, and returns the list of top k results based on raw counts model.
3. getHighestScoringPhrase : This API takes as input a String of words and returns a String of words including interpolated terms that give the highest co-occurence based on Dice's coefficient.
4. calculateAllStats : This API in turn calls the calculateSceneStats and calculatePlayStats to return the statistics details asked for in the assignment.

The inverted lists are stored with following names :

- Uncompressed : InvertedList.dat
- Compressed : InvertedListCompressed.dat

Design TradeOffs

I had to take the following design tradeOffs:

1. Use of json.simple library : I started off with the use of json.simple library for file extraction and pre-processing. In the course of the project, I learnt about the jackson library which makes data-binding more easier. So, I used the jackson library for the latter half of the project which involves reading from the disk and other pre-processing. Due to time constraints and design persaviness, I restrained from migrating earlier design of using json.simple library to jackson.
2. Dice's Coefficient calculation : There was a lot of flexibility over the approach for the calculation of dice's coefficient. It could be calculated before the retrieval process where the dice's coefficient for every pair of words is evaluated and a map to the highest scoring word is built and stored. One other approach was to calculate the dice's coefficient at run-time only for the terms in the query. I took the second approach. This could be a design trade-off. This approach impedes the query retrieval time. But, now that we are calculating only for the terms of interest, we reduce the book keeping. Also, the time taken to compute the coefficients at runtime is not perceptibly huge. So, I think the latter is a more favorable approach for the requirement of this project.

Questions and solutions

There were numerous questions during the design of this project. Most of them were resolved during in-class discussions. Some of them are :

- The obvious questions of dataStructures to choose for various purposes. For all the mappings between different features, I used HashMaps since these offer O(1) average search time.
- The obvious questions of data types to choose for different features. I was considering between long and int data types initially. But this was an easy question to answer as the magnitude that we are dealing with locally hardly seldom requires us to use long.
- The bigger question on the approach to choose for calculating dice's coefficient. After thorough analysis of the pros and cons and keeping in mind the resource limitations, I chose to calculate dice's coefficient at run-time. Although, this may be costly on the memory, I think it is faster to design.
- The question on which approach to use for retrieval : Document-At-A-Time or Term-At-A-Time. This was resolved in class.
- Apart from these, there were few clarification questions which were resolved by the professor.

Counts : Misleading Features

Raw Count can definitely be an inferior feature for comparing different scenes. It does not take into account the structure of the query and leaves out a lot of relevance properties. For example, consider the query "high wall". Here, the word "high" is associated with "wall". User might wish to see the results which are relevant to a high wall. But if there is a document that talks about high flying birds and contains a high frequency of "high" in it, and on the other hand a document that mentions "high wall" say twice or thrice in the document. In this scenario, the user expects to see the second document upon retrieval. But, a raw count model would retrieve the first document which is totally irrelevant.
This could be fixed in a number of ways. One of them is query expansion where we look for synonymous words and retrieve based on context of the query. This could still perform better than the raw counts model. We could look for the raw count for the entire query window as opposed to each word in the query. Although, this will work only if we have an extremely huge database of documents , thereby, increasing the probability of finding relevant matches. Further, we could use metrics to measure relevance of documents to the query. There is ongoing intensive research in this area and needs further exploring.

Answers to Statistics questions

1. The average length of the scene is : **1199.5562** words
2. The shortest scene is : **antony_and_cleopatra:2.8**
3. The longest play is : **hamlet**
4. The shortest play is : **comedy_of_errors**

Experimental Results and Compression Hypothesis

The project was built and run. The following observations were made :

1. The number of documents created : **748**. This matched the number of scenes in the original json file that was given to us. This is as expected per our my design.
2. The vocabulary size was found to be : **15635**. Vocabulary size is defined to be the number of words in all the documents.
3. Size of inverted list:

- Uncompressed : 5938 kb
- Compressed : 1919 kb

4. Time to write Inverted List to File:
   - Uncompressed : 8994 ms
   - Compressed : 933 ms
5. Time to retrieve 100-set 7 one word phrases:
   - Uncompressed : 31ms
   - Compressed : 15ms
6. Time to retrieve 100-set 7 two word phrases:
   - Uncompressed : 51ms
   - Compressed : 47ms
7. Time to retrieve 1000-set 7 two word phrases:
   - Uncompressed : 118 ms
   - Compressed : 111 ms
8. Time to retrieve 1000-set 7 two word phrases:
   - Uncompressed : 490 ms
   - Compressed : 456 ms
9. Total Time for evaluation :
   - Uncompressed : 3498 ms
   - Compressed : 1759 ms
10. Total Time for the whole experiment :
    - Uncompressed : 14716 ms
    - Compressed : 4470 ms

    It is seen that for a majority of the processes, compression works really well both space and time constraint wise. Although, it is observed that for the retrieval of 100 set of 7 word phrases, the time difference between uncompressed and compressed is not glaringly large. This could be because the data to be retrieved is too less for compression advantages to kick in. To put things into perspective, I ran an experiment with 1000 sets instead of 100 and observed that, although not a huge difference, but compression mode is definitely faster than uncompressed mode. So, *I would say that the compression hypothesis holds.* My expectation is that with larger and larger set of queries, the difference will definitely be more evident.