COMPSCI 546: Assignment 1

ppachpute@umass.edu

October 2019

1 Overview

1.1 System Description

- This repository provides APIs of retrieval from Shakespeare's collection of plays.
- Even though this API works on Shakespeare's dataset most of its components are modularized so that those can be changed easily to be ready for other datasets. This modularization also makes it easy to swap or replace components. Adding another compression method is easy, just add the compressor and update the "CompressorFactory" class.
- Given a word from one of the scenes it can calculate Dice's coefficient. That is it can find a word whose co-occurrence with the given word is highest.
- Given some set of query terms, API can search through the corpus of scenes and find out which scenes are most likely to contain those terms. The facility to store the inverted index on a file system is also provided.
- It can be stored as-is (uncompressed) in binary format or it can be compressed and then stored.

1.2 Design Tradeoffs

1. Size of byte buffer:
    - Should the byte buffer used for writing an index to file be of some large constant size (like 10 MB) or should we calculate its size by actually going through the entire inverted index and then allocating only the bytes required? Initially, I had allotted some constant size but then changed that to use only the bytes required so that the system is space-efficient.
2. Inverted Index implementation:

- Initially, I had created an inverted index by just creating a type of HashMap<String, PostingList>(something like typedef in c). However, later I added Map object within that class and then exposed some of the API methods like “getLengthOfScenes”. I Also created ”IndexerUtils” for utility-related functions like “compare” i.e. comparing two indexes, “getPlayWordCountMap” map from play name to count of words, etc.

3. Compression:

   - In the case of delta encoding, to avoid using extra space for IntegerBuffer, I stored delta encodings in place within the Inverted Index itself. By this when delta encoding is called on the inverted index, the index itself is modified (For retrieving original index I have also provided the decode method). This delta encoded inverted index is then passed through a v-byte compressor whose output is stored in a byte buffer.

4. Dice’s coefficient:

   - For now, I have not used skip pointers or any of the optimization techniques for calculating dice’s coefficient. It just loops over all the documents and checks if the current posting is for the same doc id.

2 Excercise Questions

1. Why might counts be misleading features for comparing different scenes? How could you fix this?

   - scores computed from count features do not take into consideration the position in which the words occurred. For example, consider a query “cricket bat” if we use count features then it is possible that a document containing information about insects and animals would also get a high score because those contain words “cricket” and “bat”. However, the user here was more interested in the bat used to play sport cricket. A similar example can be found in the case of our dataset of scenes.
   - Fix: Use positional information to calculate scores. This would mean score calculation is a function of positions of query terms. We can use n-gram models to evaluate the exact score.

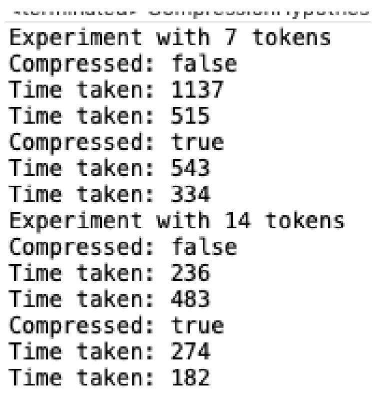
2. What is the average length of a scene? What is the shortest scene? What is the longest play? The shortest play?

   - Average length of scene: 1199.5561497326203
   - shortest scene: antony_and_cleopatra:2.8=47
   - shortest play: comedy_of_errors=16415

- longest play: hamlet=32867

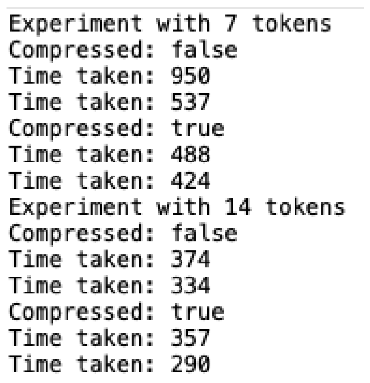
3. Does the compression hypothesis hold? What conditions might change the results of the experiment, and how would you expect them to change?

   - The compression hypothesis holds.
   - If we change the disk drive then results would change. If we use Hard disk drive instead of SSD then we can expect the un-compressed version's time to go up. For SSD the difference is visible but it is not significant



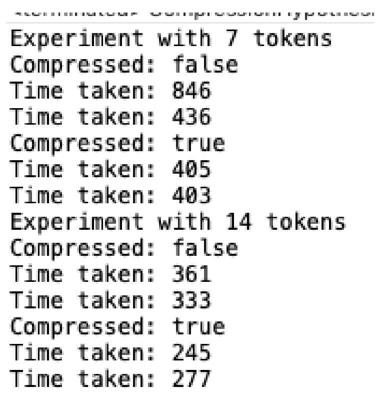
```
<terminated> CompressionHypothes
Experiment with 7 tokens
Compressed: false
Time taken: 1137
Time taken: 515
Compressed: true
Time taken: 543
Time taken: 334
Experiment with 14 tokens
Compressed: false
Time taken: 236
Time taken: 483
Compressed: true
Time taken: 274
Time taken: 182
```

Figure 1: Result 1



```
Experiment with 7 tokens
Compressed: false
Time taken: 950
Time taken: 537
Compressed: true
Time taken: 488
Time taken: 424
Experiment with 14 tokens
Compressed: false
Time taken: 374
Time taken: 334
Compressed: true
Time taken: 357
Time taken: 290
```

Figure 2: Result 2



```
Experiment with 7 tokens
Compressed: false
Time taken: 846
Time taken: 436
Compressed: true
Time taken: 405
Time taken: 403
Experiment with 14 tokens
Compressed: false
Time taken: 361
Time taken: 333
Compressed: true
Time taken: 245
Time taken: 277
```

Figure 3: Result 3

3 Classes for evaluation questions

1. Compare the vocabulary of the two indexes (terms and counts) to ensure they are identical.
    - CompareTwoIndexes
2. Randomly select 7 terms from the vocabulary. Record the selected terms, their term frequency and document frequency. Do this 100 times.
    - Random7TermsAndStatistics
3. Using Dice's coefficient (see section 6.2.1 and page 201), identify the highest scoring two word phrase for each of the 7 terms in your set of 100.
    - CalculateDice
4. Using your Retrieval API and the 100 sets of 7 terms as your queries, perform a timing experiment to examine the compression hypothesis. Repeat the experiment using the 100 sets of 7 two 14 word phrases terms.
    - CompressionHypothesisValidation