Programming Assignment 1 - Indexer Report

Submitted By: Ankita Mehta

October 6 , 2017

1 Indexer and Retrieval API

Solution:

1.1 Indexer Design:

Indexer has been created on a small dataset of shakespeare plays using Python language. Two types of indices have been created - compressed and uncompressed. You can call the wrapper ***"main.py"*** to create different indices by passing different command line arguments. Different scenes have been considered as different documents while creating indexes. This wrapper also creates the dice coefficient(Given the word **a**, it finds the most common word **b** within the window of 2 i.e ab or ba) of 100 random queries , where each query has 7 random words from the vocabulary. This wrapper takes into account 4 command line arguments:

1. ***-d or −−json_data_path*** : It is the mandatory argument, which specifies the json data path.
2. ***-u or −−uncompressed*** : It specifies the uncompressed index.
3. ***-c or −−compressed*** : It specifies the compressed index.
4. ***-dc or −−dcoefficient*** : It specifies the dice coefficient.

Note: When no argument is chosen from -u , -c , -dc - the wrapper will create the compressed index with no dice coefficient.

uncompressed Index:

Various steps followed to create the uncompressed index:

1. Firstly, the wrapper will process the JSON file and create various mappings i.e, docNo - to - sceneId, sceneId - to - docNo, docNo - to - playId, docNo - to - sceneLength, docNo - to - data. Also, it stars numbering the documents from 1 rather than 0 and at 0th docNo in docNo_sceneId_mapping, it stores the total number of documents it processed.
2. After creating all mappings, it will create the postings ("create_positional_index" function written in indices_creation.py code) document by document for every term by using the number of documents that it stored in the previous step at 0th index in docNo_sceneId_mapping. It makes a dictionary positional_index of format : $term : [list\_of\_postings]$, where each posting is a list of of format: $[docNo, count, [list\_of\_positions\_in\_docNo]]$
3. After having the postings, it will save them as a list of binary numbers in the binary file. While doing this, it will create a lookup table which is a dictionary of the format : "term: position, length, document_frequency, collection_term_frequency"
4. Now, it will save 5 mappings to disk at the location "Assignment1/results/uncompressed/" . At this point, it will also save the manifest file having the paths of all data structures dumped while making the index at the location : "Assignment1/unc_manifest"
5. Having done this, it will find the average length of scene, shortest scene, longest play and shortest play . Then it will append these information to the manifest file created in the previous step.

compressed Index:

Various steps followed to create the compressed index:

1. Step 1 is same as above : create the mappings
2. Step 2 is also same as above : create the postings of every term document by document
3. Now it will delta encode the whole list of every term. After having the delta encoded postings, it will perform VByte encoding and save them as a list of binary numbers in the binary file. While converting to binary numbers, it will create a lookup table which is a dictionary of the format : "term: position, length, document_frequency, collection_term_frequency"
4. Now, it will save 5 mappings to disk at the location "Assignment1/results/compressed/" . At this point, it will also save the manifest file having the paths of all data structures dumped while making the index at the location : "Assignment1/comp_manifest"
5. Having done this, it will find the average length of scene, shortest scene, longest play and shortest play . Then it will append these information to the manifest file created in the previous step.

Nearest Word:

Following steps followed to find the Nearest word:

1. After creating the indices either uncompressed or compressed, according to the user's choice, it will generate 100 queries, where each query is a set of 7 random but unique terms.
2. After generating the queries, it will save them in the file "Assignment1/oneword_query.txt" and then for every term in the query, it will find the most frequent word occurring in the window size of 2 (using the highest dice coefficient score). It will save the word and its nearest word in the file "Assignment1/twoword_query.txt".

   Note: Format of "Assignment1/oneword_query.txt" is - set of 100 lines. Where each line is a set of 7 terms.
   Format of "Assignment1/twoword_query.txt" is - set of 100 lines. Where each line is a set of 14 terms.

1.2 API Retrieval Design:

After creating compressed, uncompressed indices and two query files , RetrievalAPI will check the compression hypothesis. It will generate the document scores using Document_at_A_time_Retrieval method. It will give you the score for every document in which at least one of the query word is occurring. This wrapper takes into account two command line arguments:

1. ***-q or −−query_path*** : It is the mandatory argument, which specifies path to query file.
2. ***-m or −−manifest_file_path*** : It is also a mandatory argument, which specifies path to manifest file generated while creating indexes. This file will define whether Retrieval has to be done for compressed or uncompressed file.

1.3 Design Tradeoffs:

1. JSON file that the indexer is supposed to process should be in a specific format i.e it is a dictionary in which every key has values as a list of dictionaries. The internal dictionary should have atleast 3 keys : 'playId' , 'sceneId' , 'text'.
2. Information about the scenes and plays have been written in the manifest file later on i.e. after the name of all auxiliary structures have been written in the manifest file. This has been done just so that in case, we need to add another information that can be added later on.
3. Vocabulary has been accessed twice while we compute the nearest word. This has been done because, while we generate the random words, it will update the vocabulary every time. So that it shouldn't generate the repeated words. 2nd time, vocabulary is accessed to find the dice coefficient of those randomly selected words.

4. Also, while the random words have been selected, their collection term frequency and document frequency have not been stored separately. This has been accessed through the API dynamically.

5. In retrieval API, a copy of Inverted lists of query words have been made for calculating the doc score. This has been done because, rather than taking pointers for every list and maintaining them, the temp set of inverted lists has been updated on the go.

6. In retrieval API, the document scores have been stored as negative numbers. Because in Python, in priority queue the element with minimum value comes first. So, to retrieve the document with maximum score at first, document scores have to be stored as negative integers.

1.4 Various Questions and how they were approached

Various difficulties that were faced while doing the project and how they were approached:

1. The very first problem that occurred was the size of my compressed inverted list was larger than the uncompressed one. Then i figured out that I was storing the binary numbers as a text file.

2. So, the next issue came was how to store the numbers in a binary file (*This was really a pain at first using python.) . I solved this issue by playing with struct.pack() library and figured out how I can store the numbers into a binary file.

3. There were many small issues which came while I proceed with the project like : what was the second word in the second query list is , which API retrieval method to implement. Then I figured out those after having discussion with the professor.

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

Solution:

Counts can be the misleading features for comparing different scenes, because of following reason:

1. In pre-processing step we didn't remove the stopwords. So there could be a case where length of scene has increased because of the stopwords. This can be fixed by removing the stop words from all the scenes and then compare them.

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

Solution:

1. Average length of scene is: 1199.55614973

2. Shortest scene is: "$antony\_and\_cleopatra$: 2.8"

3. Longest play is: *"hamlet"* of length 32867

4. Shortest play is: *"comedy_of_errors"* of length 16415

4 Compression Hypothesis

Solution:
Experimentation was performed on compressed and uncompressed index using one word query and two word query files.

1. With uncompressed and one-word query file , on an average it took 0.00833788 seconds.

2. With compressed and one-word query file , on an average it took 0.00921255 seconds.

3. With uncompressed and two-word query file , on an average it took 0.03398347 seconds.

4. With compressed and two-word query file , on an average it took 0.04303494 seconds.

From above results, it can be seen that compressed one is taking a little more time comparatively to the uncompressed index. So, the compression hypothesis doesn't hold with this index. This is happening because the index has been created on a very small dataset. Given the large dataset, reading small numbers + decoding is expected to take lesser time than reading the large numbers.