COL764 Assignment 1 Report

Kshitij Alwadhi (2019EE10577)

30th August 2021

1 Introduction

In this assignment, we implemented a Boolean Retrieval Model for Information Retrieval. An efficient-to-query Inverted Index was created on-disk to save RAM usage.

2 Implementation Details

2.1 Pre-Processing the Data

Multiple steps are involved due to the structure of the data. The data given to us is in the form of XML files. To process this, we use the BeautifulSoup library and parsed the content using 'html.parser' which is fed as an argument to BeautifulSoup while reading the contents.

After this step, the goal is to clean the data from the punction marks /symbols and then stem the words. We use PorterStemmer and Regex Matching for this.

- **Parsing:** After reading the file contents using BeautifulSoup, we need to extract the required fields. We are given given the XML tags that we need to process as an argument while the python program is called. In that file, the first word indicates our DOCID and the rest of the lines are the fields in which our text content is present on which we need to build our inverted index. We extract the required fields using the HTML like tree structure that we get from BeautifulSoup.
- **Regex Cleaning:** Next step is that we need to clean the text document from the unwanted punctuation marks. To do this, we use Regex pattern matching and split the text accordingly.
  The following delimiters are used: `[ ',(){}.:;"‘\n]`.
- **Stemming:** After cleaning our text, we also need to stem the words. By stemming, what is meant here is reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. For example, the words "African" and "Africans" both stem to "african". For performing stemming, we use PorterStemmer.
- **Stopwords:** We are also given a list of stopwords which we store in a set and while looping through our documents, we ommit these words.

2.2 Storing the data

Since creating an inverted index of the all the files at once is not possible due to RAM limitations, we break the directory into chunks of 100 files and process them separately. The three step process is described ahead:

2.2.1 Creating Inverted Index

First, we iterate through the documents and create a dictionary which maps a token to the list of document IDs in which it is present. A dictionary was chosen here since the lookup procedures are O(1) here. So first the data is stored in the format:

```
{<doc-id 1>: [<token 1>, <token 2>...], <doc-id 2>: [<token 1>, <token 2>...]...}
```

Next, we also create a mapping which we map doc-ID to integers since our doc-ID are alphanumeric and are difficult to encode. This is stored in the following format:

```
{<doc-id 1>: 1, <doc-id 2>: 2, ...}
```

Now, we move on to the process of creating the inverted index, in which we create a dictionary containing the tokens as the key and the value paired with it is a posting list which contains our document identifiers. This is stored in the following format:

```
{<token 1>: [posting list 1], <token 2>: [posting list 2], ...}
```

All the posting lists in this inverted list are then gap encoded to decrease the data size.

2.2.2 Dumping this inverted index into a temporary file

The inverted index of this subset of files is then dumped onto our disk using C2 encoding (analysis on this in later sections). This process also returns a dictionary which is used along with the binary file created for retrieval later on. We also maintain a vocabulary while processing the chunks of directory. After we have dumped the inverted index files for all the chunks created from the directory, we move on to the step of merging the inverted index.

2.2.3 Merging the inverted index files created

Now in this step, we loop over the words of the vocabulary. Now for each word, we look into each of the temporary inverted index files created and keep extending the postings list for that word. Now this posting list of that word is dumped into the required inverted index file using the desired compression model. More about the encoding and decoding algorithms in the next section. At a time, only one posting list is in memory which will fit in the RAM hence we have avoided the problem of inverted index overflowing the RAM. Psuedo code for this algo is below:

Algorithm 1 Merge Inverted Index

```
1:  procedure MERGE
2:      dictionary ← emptydictionary
3:      alldicts ← loadDictFiles()
4:      allidx ← loadBinFiles()
5:      for w in vocabulary do
6:          postings ← []
7:          for i, tempdict in enumerate(alldicts) do
8:              if w in tempdict then
9:                  reader ← allidx[i]
10:                 postings.extend(getPosting(reader, w, alldicts[i]))
11:             end if
12:             <do encoding and write>
13:         end for
14:     end for
15:     return dictionary
16: end procedure
```

2.3 Encoding

In this assignment, we explore various encoding techniques. One thing that is consistent across all the compression techniques is that we are storing two files at the end of this process. One is JSON dictionary dump in which we are storing a nested dictionary which contains the compression type, the document identifier to document ID map and another mapping which maps the words of the vocabulary to a vector which contains the information needed to decode the posting list depending on compression type. The other is a binary file which contains the encoded postings list of the inverted index.

2.3.1 C0 Encoding

In C0, no compression is performed, instead, each number of the posting list is represented as a 32 bit (4 bytes) number. For those numbers whose binary representation has less than 32 bits, 0's are appended at the front to make it a 32 bit number. For each word in our vocabulary, we also store the offset and the length of the posting list in our .dict file. The dictionary being stored in our .dict file for C0 encoding has the following structure.

```
{'compression': 0, 'int_to_docID': {'1':'AP880212-0001',..},
'dictionary': {<token 1>: [<offset>, <length>], <token 2>: [<offset>, <length>], ...} }
```

2.3.2 C1 Encoding

Compression c1: Works with chunks of binary data as the data in computer memory itself follows this representation.

encoding: split the $bin(x)$ in chunks of 7 bits, and each byte that is to be stored now contains these 7 bits in lsb with MSB set to 1 in all bytes except last one(least siginficant byte). The data in the first byte is padded with zeros to left if needed.

example: consider $x = 111119$ in base-10, whose binary representation is

$$bin(x) = 1\ 1011\ 0010\ 0000\ 1111$$

then $encoding(111119)$ generates three bytes with

$$10000110\ \ 11100100\ \ 00001111$$

decoding: simply read until the byte with 0 MSB and process all 7 lsb bits of read bytes accordingly.

Note that we use Gap encoding with $c1$.

In C1, we are doing a modified VB decoding on the posting list. Since after VB decoding, we get chunks of 8 bits (1 byte) for every number, we can push them in a bytesarray to our binary file without worrying about the padding. Here as well, for each word in our vocabulary, we also store the offset and the length of the posting list in our .dict file. The dictionary being stored in our .dict file for C1 encoding has the following structure.

```
{'compression': 1, 'int_to_docID': {'1':'AP880212-0001',..},
'dictionary': {<token 1>: [<offset>, <length>], <token 2>: [<offset>, <length>], ...} }
```

2.3.3 C2 Encoding

Compression c2: Encodes each integer with $O(\log x)$ bits and thus gets close to optimal number of bits for each integer in its compression. Let,
$lsb(a, b)$ denote *b least significant bits (lsb)* of the binary representation of the number a,
$l(x) = length(bin(x))$ and
$U(l)$ denote the unary representation of a number n. Unary representation of a number n is simply $(n-1)$ 1's followed by a zero, i.e., $U(1) = 0, U(2) = 10, U(3) = 110, ...$

Given these, we define the encoding and decoding in $c2$ as follows:

encoding:

$$U\left(l\left(l\left(x\right)\right)\right) \odot lsb\left(l\left(x\right), l\left(l\left(x\right)\right) - 1\right) \odot lsb\left(x, l\left(x\right) - 1\right)$$

The symbol $\odot$ just denotes concatenation of bits.

example: For x = 119, $bin(119) = 111\ 0111$, $l(119) = 7$, $l(l(119)) = 3$, $U(3) = 110$, $lsb(7, 2) = 11$, $lsb(119, 6) = 11\ 0111$

$$\textit{encoding(119)} = 110\ 1111\ 0111$$

decoding: Read the unary code first to find $l(l(x))$ and then the next $l(l(x)) - 1$ bits to find $l(x)$, finally read next $l(x) - 1$ bits to find x. Don't forget to append 1 at the *most significant bit* (msb) to the read bits while finding the actual integers $l(x)$ and x.

Following the algorithm described in C2 in the problem statement, we observe that the length of the output of C2 encoding of a number is not necessarily a multiple of 8. To account for this, we do C2 encoding of each number in the posting list of a particular term and them append them one after the other. Now the length of this string is not necessarily a multiple of 8 so what we do is we append 0's at the start of the string formed and store the number of 0's appended in a variable called skipbits since this will be necessary at the time of decoding. For each word in our vocabulary, we store the offset the length of the posting list and the number of bits to be skipped in our .dict file. The dictionary being stored in our .dict file for C2 encoding has the following structure.

```
{'compression': 2, 'int_to_docID': {'1':'AP880212-0001',..},
'dictionary': {<token 1>: [<offset>, <length>, <skipbits>], <token 2>: [<offset>,
<length>, <skipbits>], ...} }
```

At the time of decoding, we skip the 'skipbits' number of digits from the start and then start the decoding process for the encoded string.

2.3.4 C3 Encoding

Compression c3: This compression is different from the previous two because it compresses each postings list entirely using Google's fast general-purpose compression library called *snappy* (https://github.com/google/snappy). Python bindings are available from python-snappy (https://github.com/andrix/python-snappy). Note that snappy being a general-purpose compression library, it simply takes a sequence of bytes as input and generates another sequence of bytes (compressed). It is up to you to suitably represent the sequence of numbers in the (gap-encoded) postings list as a string and feed it to snappy compress/decompress functions.

encoding: Given a postings list p consisting of gap-encoded document identifiers, compress it using snappy library.

decoding: Each time postings-list is required, bring its compressed version to memory, decompress it, and use the resulting decompressed version. You can assume that after decompression the result fits in memory.

In C3 encoding, we make use of the snappy library provided by Google. Snappy takes a string input and encodes it into binary format. So what we do here is, we take our postings list (an array), and comma separate each number present in it. We store the length of this string 'length' and keep increasing our offset by this number. Here as well, for each word in our vocabulary, we also store the offset and the length of the posting list in our .dict file. The dictionary being stored in our .dict file for C3 encoding has the following structure.

```
{'compression': 3, 'int_to_docID': {'1':'AP880212-0001',..},
'dictionary': {<token 1>: [<offset>, <length>], <token 2>: [<offset>, <length>], ...} }
```

In the end, we join all these strings into one single string and use snappy.compress method for encoding this into a binary file.

2.3.5 C4 Encoding

Compression c4: considers how flexibility in parameters can help in compression. For integer x and parameter $b = 2^k$ with $k > 0$, consider $q = \lfloor (x-1)/b \rfloor$ and $r = x - q * b - 1$, $C(r) = bin(r)$ is to represent integer r with k bits in binary, $U(l)$ is unary representation of l. The optimal value for k is based on trade off between bits required for unary representation and $C(r)$ for each x

encoding: $U(q+1).C(r)$
Either some fixed number of bits, over the entire index, can be used before each encoded integer to encode the k or chose previous integer compression methods c1 or c2.

example: consider $x = 119$, with say $k = 6, b = 64$, then $q = 1, U(2) = 10, r = 54, C(54) = 11\ 0110$.

$$encoding(119) = 1011\ 0110$$

decoding: read the unary prefix to find q, then read next k bits to find r to finally get back x

The C4 decoding depends on the value of k chosen. From trying out various values of k, we realise that the shape of length of string vs k graph is somewhat parabolic with a minima occuring in the range of $[\lfloor log_2 k \rfloor - 4, \lfloor log_2 k \rfloor - 1]$. Moreover, we also append the C2 encoding of k in the beginning of the result from C4 encoding of the postings list. Apart from this, to make the length a multiple of 8, we also append 0's at the end. After doing all this for the four values of k, we pick that value of k for which the length of the resultant string is minimum.

This way, we are doing the best possible compression by sacrificing a small amount of extra compression time. Here, for each word in our vocabulary, we store the offset and the 'numbits', which is the actual length of the of the encoded postings list before 0's were appended, in our .dict file. The dictionary being stored in our .dict file for C4 encoding has the following structure.

```
{'compression': 4, 'int_to_docID': {'1':'AP880212-0001',..},
'dictionary':{<token 1>:[<offset>, <numbits>],<token 2>:[<offset>, <numbits>],..}}
```

2.4 Retrieval

For performing the task of retrieval, we first load the .dict file into our memory. From this, we can get the following information:

1. Compression Type
2. $intToDocID$ dictionary
3. Dictionary containing the term to [offset, ...] mapping

Now whenever we have to answer a query, we first perform regex split and then stem the words given in our query. Then we need to retrieve the postings lists which is specific to each compression type.

2.4.1 C0 Decompression

From the dictionary, we get the start and size of the posting list corresponding to that particular term. Then we get the binary data from the .idx file for this postings list and then perform C0 decoding on it to get back the postings list.

2.4.2 C1 Decompression

This is also similar to C0. From the dictionary, we get the start and size of the posting list corresponding to that particular term. Then we get the binary data from the .idx file for this postings list and then perform C1 decoding on it to get back the postings list.

2.4.3 C2 Decompression

Here from the dictionary, we get the start, size of the posting list and the number of skipbits corresponding to that particular term. Then we get the binary data from the .idx file for this postings list and then perform C2 decoding on it to get back the postings list. The number 'skipbits' helps us decide how many 0's to skip at the front.

2.4.4 C3 Decompression

This is different from the other decompressions in one key aspect. Here, we need to first perform snappy's decompress method on the whole file to get back the string. Then we get the start and size of the posting list from the dictionary corresponding to our particular term and using that we get our postings list.

2.4.5 C4 Decompression

Here, from the dictionary, we get the start and the 'numbits' from the dictionary corresponding to our term. Now first, we get the value of 'k' by performing a single C2 decode to get the first number. We also store the jumps made by our iterator while performing this C2 decode on our substring. Now for the next '$numbits - jumps'$ we perform C4 Decode using our k that we got to get our postings list.

Now we have our list of postings list from which we get our intersection by the following algorithm discussed in lecture. We map the elements of this intersection back to their original document ID and that is the answer to our query.

3 Performance

For comparing the performance of the different compression techniques, we compute the following metrics:

3.1 Index Size Ratio (ISR)

$$ISR = \frac{|D| + |P|}{|C|}$$

where $|D|$ is the size of the dictionary file, $|P|$ is the size of the postings file, and $|C|$ is the size of the entire collection.

	IDX	**DICT**	**ISR**
C0	14,72,12,312	1,27,08,892	0.310348866
C1	4,18,16,067	1,25,12,980	0.105432911
C2	3,20,18,705	1,34,01,383	0.088143864
C3	5,42,09,049	1,26,41,048	0.129731713
C4	2,65,48,770	1,27,51,427	0.076267382

Table 1: All the sizes are reported in bytes and the total size of our collection is 51,52,94,952 bytes.

In decreasing order of performance while comparing the the compression, we notice that:

$$C4 > C2 > C1 > C3 > C0$$

3.2 Compression Speed

This is reported as the time taken in seconds on my system during compression (using C0, C1, C2, C3, C4).

	Total Time
C0	870.9356
C1	873.2596
C2	866.3376
C3	788.2062
C4	1076.813

It's not possible to directly compare the results with C0 (supposedly no compression there) since the I/O takes a considerably longer time while we are performing the write to binary file in C0 because of increase in number of bits.

3.3 Query Speed

This is reported as the Average time taken per query in μ - seconds per query.

$$QuerySpeed = \frac{|T_Q|}{Q}$$

where T_Q is the time taken for answering (and writing the results to the file) for all given queries, and Q is the number of queries.

	Total Time	**Queries**	**Avg Time per Query**
C0	42.99954605	50	0.859990921
C1	9.849957705	50	0.196999154
C2	8.947190762	50	0.178943815
C3	2.690040588	50	0.053800812
C4	7.15001297	50	0.143000259

Table 2: All time reported in seconds