Information Retrieval and Web Search

COL764

Assignment 1

Japneet Singh
2019MT10696

Semester I 2021-22

Contents

1 Introduction

This assignment implements a Boolean model for Information Retrieval. The IR system developed supports the following operations:

Single keyword retrieval: Returns all document-ids whose document content contains the specified keyword.
Multi keyword retrieval: Returns all document-ids whose document content contains all the specified keywords.

A stemmed term is a a single word, suitably stemmed to get the root word if it is in its derived form.
A document is treated as a set of words, built by collecting and stemming every word contained by it.
A term and a document are said to match if the stemmed term appears in the document.

2 Implementation and Tuning Details

2.1 Pre-Processing the Text

- We maintain the following data structures:
  - a dictionary mapping docId to its DOCNO. The docId are assigned in numerical order of parsing the document. DOCNO is the unique identifying string enclosed within <DOCNO>...</DOCNO> for each tag
  - a Postings dictionary, mapping each keyword to its gap-encoded postings list.
  - a lastDoc dictionary, mapping each keyword to the docId of the document it was last found in. The image of a keyword defaults to 0, signifying it is a new word. If such a keyword is encountered, it is added to the vocabulary.
  - an offsetAndLength dictionary, which will be finally written to the <indexfile>.dict. It maps each keyword to the offset and length of its postings list in the inverted index generated in the end.

  Python dicts are used as they provide O(1) average tie complexity for lookup and are efficienct even while dealing with large amounts of data.

- The XML file is parsed using the Beautiful Soup library and the *html.parser* . *lxml*, although slightly faster in practice was not used as the collection does not contain a single XML document per file. Each file has multiple root <DOC> tags and *lxml* only recognises the first one. Additionally, *lxml* also removes <HEAD> tags.

- On parsing, all the documents in a single file (enclosed in <DOC>..</DOC> are inserted into a list. This helps us as the relevant XML tags are not fixed and each tag can occur varying number of times (in fact, may not occur at all) in each document.
- One by one, the text enclosed in required tags is extracted. Stop words are removed in this step. For splitting, the delimiters used are -

  ```
  ,.:;"(){}[]\n‘’
  ```

  Occurrences of these delimiters are replaced by a single space (and later split by white space) using Python 3.7's string maketrans() and translate() which uses C level lookup resulting in faster replacements as compared to replace() function.

2.2 Creating the Inverted Index

- We set a feasible block size, in terms of number of files parsed, so that the sub-inverted index created for that block fits in the memory. For example, n/10 where n is the total number of files. On reaching this blocksize, we write the postings lists created by then to helper files on-disk in the within the same directory. Once these files are created, the postings list and the sub-dictionary created are cleared.
  The lastDoc dictionary and the vocabulary/offsetAndLength dictionary stay in memory. This allows us to continue the postings list for a term with gaps adjusted according to the sub-list already stored in the previous file. For example, say the <word> has the gap-encoded postings list

  ```
  2, 3, 1, 4, 10, 3, 3
  lastDoc[<word>] = 26
  ```

  This sub-list is written to the file and a new list is created for <word> in the new sub-inverted index for the next block. If the next docIds in which <word> appears are 39, 42 and 44, we simply start the next sub-list from 39-26=13

  ```
  13, 3, 2
  lastDoc[<word>] = 44
  ```

  The advantage of this being we can simply extract these lists and append them, in order ans write the full list to <indexfile>.idx while merging the block sub-indices.

- Once all the files are parsed and postings lists created, we merge the sub-indices. This is done by creating file objects for each of the sub-index, iterating over the keys, extracting the sub-lists in order, encoding them as required and writing them finally to the resultant <indexfile>.idx For example, the sub-lists for <word> above will be concatenated as

  ```
  <word> -> 2, 3, 1, 4, 10, 3, 3, 13, 3, 2
  ```

- The dictionary with pointers for each keyword to its inverted index list is dumped as 'UTF-8' encoded JSON strings to <indexfile>.dict . JSON is easily and quickly converted into python dictionaries, helping in efficient and quick retrieval.

2.3 Compression Methods

2.3.1 C0 - No compression

The posting lists are converted into strings of comma separated integers. The resultant strings are encoded in 'UTF-8' and written to <indexfile>.idx .

2.3.2 C1 - Variable Byte Encoding

Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are payload and encode part of the gap. The first bit of the byte is a continuation bit . It is set to 0 for the last byte of the encoded gap and to 1 otherwise.

```
def c1_encode(n):

INPUT: an integer, n
OUTPUT: an array of decimal numbers, each being the decimal
equivalent of the 8bits of the resultant encoding, in order

    byte_arr = []
    byte_arr.append(n%128)
    n = n//128
    while(True):
        if(n==0):
            break
        else:
            byte_arr.insert(0, n%128 + 128)
```

```
            n = n//128
        return byte_arr

    c1_encode(111119)
    VBE of 111119 = 10000110 11100100 00001111
    Resultant array = [134, 228, 15]

    def c2_decode(s):

    INPUT: a string s, Variable Byte Encoded
    OUTPUT: an integer n, whose VBE is s

        decoded = 0
        totalBytesDecoded = 0
        while(totalBytesDecoded<len(s)):
            byte = (read next bytes i.e. next 8 binary digits)
            totalBytesDecoded += 1
            readByte = (decimal form of byte)
            if(readByte<128):
                decoded = decoded*128 + readByte
                return decoded
            else:
                decoded = decoded*128 + (readByte-128)
```

2.3.3 C2

Encoding of x := U (l (l (x))) $\odot$ lsb (l (x) , l (l (x)) $-$ 1) $\odot$ lsb (x, l (x) $-$ 1) 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

```
    def c2_encode(x):
    INPUT: integer n
    OUTPUT: a string representation of encoding of n

        encoded = ""
        lx = x.bit_length()
        llx = lx.bit_length()

        ->U(l(l(x)))
        for _ in range(0, llx-1):
            encoded += '1'
        encoded += '0'

        ->lsb (l (x) , l (l (x)) − 1)
        a1 = lx
        b1 = llx-1
        temp1 = ""
        for _ in range(0,b1):
            temp1 = str(a1%2) + temp1
            a1 = a1//2

        ->lsb (x, l (x) − 1)
        a2 = x
        b2 = lx-1
        temp2 = ""
        for _ in range(0,b2):
            temp2 = str(a2%2) + temp2
            a2 = a2//2

        encoded += temp1
        encoded += temp2
        return encoded
```

As writing and reading from a file is only done in units of bytes, to make the length of the encoded string a multiple of 8, we pad the string code of the **complete postings list** of a keyword with required 1s.
We ensure this doesn't corrupt the encoding as while decoding, we know where a new gap's code starts.
Moreover, U(l(l(x)) will always end in a 1, if we don't find a 0 after a series of continuous 1s while reading an encoded string, we deduce that these 1s are being used for padding.

```
    def c2_decode(encodedString):
        postings_list=[]

    INPUT: encoded string of complete list, with padded 1s
    OUTPUT: Decoded Postings List

        iter = 0
        while(iter<len(encodedString)):
            current_len = 0
            while(encodedString[j]!='0'):
                current_len+=1
                iter+=1
                if(iter>=len(encodedString):
                    break
            if(iter>=len(encodedString) and encodedString[-1]==’1’):
                break
            iter+=1
            current_len+=1
            llx=current_len
            lx=1
            for _ in range(0,llx-1):
                bit = int(encodedString[j])
                lx = lx*2 + bit
                iter = iter+1
            x = 1
            for _ in range(lx-1):
                bit = int(encodedString[j])
                x = x*2 + bit
                iter = iter+1
            postings_list.append(x)

        return postings_list
```

2.3.4 C3 - Snappy

Postings list is compressed using Google’s fast general-purpose compression library called *snappy*. Each posting list is converted into a string of comma separated gaps, which are compressed using snappy’s compress() function.

2.3.5 C4 - Not Implemented

2.3.6 C5

We find the k such that most of the integers in the list can be represented using k bits per integer and then find suitable b such that these values fit in range [b, b + $2^k - 2$].
b is taken as the smallest number in the first 8/10th of the list, consequently, k becomes $[log_2(maxElementinthesaerange - b + 2)] + 1$ where [.] represents G.I.F.
Each number is shifted with respect to b to make the numbers fall in the range [0, $2^k - 2$]. The numbers which fall out (i.e., $n_i > 2^k - 2$) can be represented with separate simple list encoded in C1 scheme appended to the above encoded list.
The C5 encoded numbers and out of range numbers are separated by k-bit binary

representation of $2^k - 1$, i.e. k 1s.
Each list begins by VBE sequence of b, followed by VBE sequence of k. Next are k bit blocks of gaps, their end denoted by a block of k consecutive 1s. Next follow the out of range numbers in their VB encoding.

```
def c5_encode(postings_list pl):
INPUT: a postings list
OUTPUT: encoding of the postings list, in string form
    b = pl[0]
    maxEle = pl[0]
    k = 2
    cutOff = 0

    if(len(pl)==1):
        comp1 = ['{0:08b}'.format(x) for x in c1_encode(b)]
        comp2 = ['{0:08b}'.format(x) for x in c1_encode(k)]
        return(''.join(comp1)+''.join(comp2)+'11'*(cutOff+1))

    while(cutOff<len(pl) and (cutOff+1)*100/len(pl)<80):
        b = min(b, pl[cutOff])
        maxEle = max(maxEle, pl[cutOff])
        cutOff += 1

    if(len(pl)>1):
        k = math.floor(math.log2(maxEle-b+2)) + 1
    while(cutOff<len(pl)):
        if(pl[cutOff]<=maxEle and pl[cutOff]>=b):
            cutOff+=1
        elif(pl[cutOff]<b and math.floor(math.log2(maxEle-pl[cutOff]+2)) + 1 ==
            b=pl[cutOff]
            cutOff+=1
        elif(pl[cutOff]>maxEle and math.floor(math.log2(pl[cutOff]-b+2)) + 1 ==
            maxEle = pl[cutOff]
            cutOff+=1
        else:
            break

    format_k = '{0:0'+str(k)+'b}'
    comp1 = ''.join(['{0:08b}'.format(x) for x in c1_encode(b)])
    comp2 = ''.join(['{0:08b}'.format(x) for x in c1_encode(k)])
    comp3 = ''.join([format_k.format(x-b) for x in pl[0:cutOff]])
    if(cutOff==len(pl)):
        to_write = comp1+comp2+comp3
        padding = (8 - (len(to_write)%8))%8
        return to_write+('1'*padding)
    else:
        comp4 = [['{0:08b}'.format(x) for x in c1_encode(ele)] for ele in pl[cu
        comp4 = ''.join([''.join(x) for x in comp4])
        to_write = comp1+comp2+comp3+('1'*k)+comp4
        padding = (8 - (len(to_write)%8))%8
        return to_write+('1'*padding)
```

To decode, we first use C1 decoder to find out b and k respectively, then read blocks of k bits to recreate the list of in-range numbers, adding b to get back the original gaps. Next, once we encounter the ending marker, we again use C1 decoder to get the out-of-range numbers.

2.4 Retrieval Details

- The type of encoding is stored at the start of the invidx file using one byte, corresponding to one out of $0, 1, 2, 3, 4 and 5$.
- Once we get the compression type, we read the query file to form the list of query keywords. They are stemmed and parsed just like the documents to ensure same words reach the same stemmed forms.
- Once done, we go through the inverted index, and create a list of posting lists, reading only the required posting lists from the file using the offset pointers stored in the dictionary.
- Once this list is created, we sort it in ascending order of size of the posting lists. Starting with the smallest, we iterate through all the lists, removing docIds not found in the current list, effectively taking an intersection of all the lists - analogous to an AND boolean operation.

3 Performance and Metrics

- Block size used: 300 files each i.e. 3 subindices were created containing the inverted indices corresponding to docId 1-300, 301-600 and 601-686 respectively.

3.1 Index Size Ratio (ISR)

$ISR = (|D| + |P|)/|C|$ 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. |C| = 515,294,952 bytes

- **C0**
  |D| + |P| = 12,655,907 bytes + 97,007,491 bytes = 109,663,398 bytes
  ISR = (109,663,398)/(515,294,952) = **0.212816752**
- **C1**
  |D| + |P| = 12,580,402 bytes + 41,773,166 bytes = 54,353,568 bytes
  ISR = (54,353,568)/(515,294,952) = **0.1054804977**
- **C2**
  |D| + |P| = 12,583,086 bytes + 31,982,367 bytes = 109,663,398 bytes
  ISR = (44,565,453)/(515,294,952) = **0.08648532812**
- **C3**
  |D| + |P| = 12,666,897 bytes + 59,667,586 bytes = 72,334,482 bytes
  ISR = (72,334,482)/(515,294,952) = **0.140374909**
- **C5**
  |D| + |P| = 12,599,558 bytes + 38,054,563 bytes = 109,663,398 bytes
  ISR = (50,654,121)/(515,294,952) = **0.0983012172**

3.2 Compression Speed

The total time taken to compress all the postings lists, which is the time difference between indexing with a compression strategy and indexing without any compression.

- **C0**
  Time Taken, T_0 = 705,881.560087204 μS
- **C1**
  Time Taken, T_1 = 740,704.4847012 μS
  Compression Speed = $T_1 - T_0$ = **34,822.924614** μS
- **C2**
  Time Taken, T_1 = 806,938.8651847839 μS
  Compression Speed = $T_2 - T_0$ = **101,057.3050975799** μS
- **C3**
  Time Taken, T_1 = 713,803.0400276184 μS
  Compression Speed = $T_3 - T_0$ = **7,921.4799404144** μS

- **C5**
  Time Taken, $T_1 = 763{,}088.8228416443\ \mu S$
  Compression Speed = $T_5 - T_0$ = **57,207.2627544** μS

3.3 Query Speed

Average time taken per query (including decompression of list(s)).
Query Speed = $|T_Q|/Q$ 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.
Sample Query File given has 50 Queries => Q = 50

- **C0**
  $|T_Q| = 2957.406759262085\ \mu S$
  Query Speed = **59.1481351852** μS per query
- **C1**
  $|T_Q| = 4599.591255187988\ \mu S$
  Query Speed = **91.9918251** μS per query
- **C2**
  $|T_Q| = 10078.965902328491\ \mu S$
  Query Speed = **201.5793180466** μS per query
- **C3**
  $|T_Q| = 3058.1717491149902\ \mu S$
  Query Speed = **61.1634349823** μS per query
- **C5**
  $|T_Q| = 7724.3571281433105\ \mu S$
  Query Speed = **154.4871425629** μS per query