Assignment 1

Assignment Description

Built an efficient Boolean retrieval system for English corpora by constructing an on-disk inverted index, consisting of a dictionary file and a single file of all the postings lists. Tokenization, stop word removal, and stemming have been used to generate the indexes. The system supports single keyword query retrieval and multi-keyword query retrieval. I have also experimented with 3 different compression techniques to facilitate efficient storage and retrieval.

Input

I have indexed over an English document collection – extracted from a benchmark collection in TREC. Each document has a unique document id (DOCNO), and the overall document is represented as an XML fragment. Only the content enclosed in specific XML tags (as mentioned in the input xml-tags-info file) has been indexed.
The queries are given as input in the form of a query file.

Output

Generate a dictionary (contains the indices) and an index file (contains all the postings list). The dictionary file also contains information about the implemented DOCNO to docid mapping used, the list of stopwords, and the compression mechanism used. The index file is a binary file containing all the postings list. Finally, a results file is generated which contains each query and the corresponding document number in which it is present and the similarity score (1.0 in case of a boolean retrieval system).

Program Structure

invidx_cons.py - generates the dictionary and index file.
invidx.sh [coll-path] [indexfile] [stopwordfile] {0|1|2|3|4|5} [xml-tags-info]

General flow of the program -

1. read_xml_tags - read the xml tags to be indexed from file
2. read_stopword_file - read the list of stopwords from file
3. read_collection - reads the collection files -> parses them -> generates the inverted index in memory
4. dump_to_file - writes the inverted index to files - 1. Dictionary file 2. Postings lists Compression is performed on the postings list while writing to file.

boolsearch.py - generates the final results file
boolsearch.sh [queryfile] [resultfile] [indexfile] [dictfile]

General flow of the program -

1. load_invidx - reconstructs the inverted index into memory by reading the dictionary and index files (appropriate decompression technique is used to read the file)
2. query - reads the query file line by line and writes the result to a separate results file.

Index creation specifications

1. I have maintained a python defaultdict (similar to unordered_map in C++) for storing the inverted index and a document number to id mapping which I use for storing the postings list in memory and writing to file.
2. Process for parsing file -
    a. I read each file in the collection and extract all the text information from it.
    b. This is then passed to the parse_files() method which parses all the documents in that file. For each document in that file tokenization (splitting using the delimiters), followed by stopword filtering and then finally stemming is done to generate the index. This index is then appended to the inverted index constructed so far. The keys of the inverted index are the indexes and the values are a list of document ids (obtained from the document number to id mapping).
    c. After all the files in the collection have been parsed and the inverted index has been constructed, then it is written to disk. If a compression scheme has been specified then the appropriate encodings are called (explained in the compression section).

Result generation and query processing

1. The index and dictionary files are read to generate the in memory inverted index which is stored as a python defaultdict. If any compression has been specified in the dictionary file then an appropriate decoding method is used (explained in the compression section).
2. After loading the inverted index into memory the queries are processed.
3. Each query is broken into keywords using the same delimiters as were used in the case of tokenizing the documents in the collection, then stopword removal and stemming is performed to get the final keywords.
4. The postings list corresponding to these keywords is then obtained from the inverted index and the intersection of all the postings list will give our final resulting set of documents.

Compression Section

The dictionary is always stored in the same format not depending on the compression scheme specified.

C0 - No compression

1. For each postings list in the inverted index, I simply write the size of the postings list followed by each docid in the list (integral) into the file as a 4 bit unsigned binary number. For converting the integer to binary I use the to_bytes() method.
2. While decoding the file I first get the size of the file (bytes that are to be read). After that I read the first 4 bytes from the file to obtain the length of the first postings list, following which I read the corresponding number of bytes from the file and convert them back to integers to get the document ids in the postings list. For this I use the int.from_bytes() method.

   The below metrics have been computed using the test data that was given.

   Index size ratio (ISR) = (|D| + |P|) / |C| = (7.36 + 137) MB / 515.3 MB = 0.2801

   Query speed = |Tq| / |Q| = 37.3947 / 50 = 0.0748 secs = 74.8 ms = 747.894 us

C1 compression

1. For each postings list I first generate the gap encoded list.
2. I write the length of this postings list to file by converting to binary and storing in a python byte array. This array can then be written to file directly. The purpose of this representation is that I need not store the size of the postings list in a 4 byte integer but can be stored in a data array and 1 byte which contains the number of bits present in the binary representation of the size. For eg -
   a. If size = 100 the bin(size) = 01100100.
   b. The first byte with store - 01100100 and the next 1 byte will store the data array containing size = 100.
   c. Therefore instead of 4 bytes I have used only 2 bytes
3. For each entry in the gap encoded list I generate its binary representation and apply the variable encoding scheme as mentioned to obtain a string representation of the number. This is again stored in a byte array and written to the file.
4. While storing in the byte array I have to ensure that each binary number has been padded to a multiple of 8 bits so as to ensure I read the correct number of bits while decoding.
5. While decoding I first find the size of the index file (total number of bytes to read)

6. While decompressing I follow the steps mentioned in the document of reading till the first byte with ‘0’ as the first digit in the binary representation.
7. First I read the size of the postings list, then while I have not obtained a ‘0’ as the first bit in the binary representation I keep reading bytes from the file and storing in an array.
8. Since only the last 7 bits matter in each 8 bit encoding, I store the bits in a string which is later read from to get the gap encoded document id.

   The below metrics have been computed using the test data that was given.

   Index size ratio (ISR) = (|D| + |P|) / |C| = (7.36 + 39.5) MB / 515.3 MB = 0.091
   Compression speed = 281.292 sec = 281292 ms
   Query speed = |Tq| / |Q| = 200.0884 / 50 = 4.001768 secs = 400.1768 ms = 400176.8 us

C2 compression

1. For each postings list I first generate the gap encoded list.
2. I write the length of this postings list to file by converting to binary and storing in a python byte array. This array can then be written to file directly. The purpose of this representation is that I need not store the size of the postings list in a 4 byte integer but can be stored in a data array and 1 byte which contains the number of bits present in the binary representation of the size. For eg -
   a. If size = 100 the bin(size) = 01100100.
   b. The first byte with store - 01100100 and the next 1 byte will store the data array containing size = 100.
   c. Therefore instead of 4 bytes I have used only 2 bytes
3. For each entry in the gap encoded list I generate its binary representation and apply the encoding scheme as mentioned to obtain a string representation of the number. This is again stored in a byte array and written to the file.
4. While storing in the byte array I have to ensure that each binary number has been padded to a multiple of 8 bits so as to ensure I read the correct number of bits while decoding.
5. While decoding I first find the size of the index file (total number of bytes to read)
6. After that I read the size of the postings list. Looping over the size I read in data 1 byte at a time into a byte array.
7. Then I follow the steps as mentioned to obtain the unary encoding, from which we obtain l(l(x)) and then after reading the next l(l(x)-1) bits we obtain l(x) and the next l(x)-1 bits give us the document id.

8. Since I have stored everything using byte arrays I need to take special care while looping so as to not miss read bytes from 1 doc id to another.

   Index size ratio (ISR) = (|D| + |P|) / |C| = (7.36 + 48.9) MB / 515.3 MB = 0.109
   Compression speed = 197.2435 sec = 197243.5 ms
   Query speed = |Tq| / |Q| = 206.9120 / 50 = 4.12258 secs = 412.258 ms = 412258us

C3 compression -

1. For each postings list I first generate the gap encoded list.
2. For each gap encoded list I store it in a python long int array and ssing snappy.compress() generate a compressed byte representation to be written to file.
3. I write the length of this generated byte string to file by converting to binary and storing in a python byte array. This array can then be written to file directly. The purpose of this representation is that I need not store the length of the list in a 4 byte integer but can be stored in a data array and 1 byte which contains the number of bits present in the binary representation of the size. For eg -
    a. If len = 100 the bin(len) = 01100100.
    b. The first byte will store - 01100100 and the next 1 byte will store the data array containing len = 100.
    c. Therefore instead of 4 bytes I have used only 2 bytes
4. Finally I write the compressed string to file.
5. While decoding I first find the size of the index file (total number of bytes to read)
6. Next I read the length of the compressed string and the corresponding number of bytes. Passing this to snappy.uncompress() gives me the desired array representation.
7. From this I can generate the original postings list.

   Index size ratio (ISR) = (|D| + |P|) / |C| = (7.36 + 75.8) MB / 515.3 MB = 0.164
   Compression speed = 342.12651 sec = 342126.51 ms
   Query speed = |Tq| / |Q| = 54.4829 / 50 = 1.089658 secs = 108.9658 ms = 108965.8 us