Working and Design of the built search engine

Overview

The developed search engine is designed to obtain the results for search terms in the order of results descending order of BM25 score, for ease of inspection we have put constraints on how many search results are output however this is a cosmetic cut off and is done only during the display stage and has no impacts in the processing or ranking of the search results.

This search engine utilizes the data from the URL http://commoncrawl.org/the-data/ . I have used the offline download to generate the indexes however there is an option to also directly use the data from the Amazon cloud.

The data available in the link is in GZ format. The data used for the demo has been downloaded from the link http://commoncrawl.org/2016/10/september-2016-crawl-archive-now-available/ . Using the links available for WET files(https://commoncrawl.s3.amazonaws.com/crawl-data/CC-MAIN-2016-40/wet.paths.gz) the data is downloaded into a chosen file path.

This data is then used by the index generation programs to first convert the data which is again already available in the form where the HTML tags are already stripped from the data dump. Here the basic concept employed in generation of the inverted index is stripping all the words available for every HTML page, associate the words with the URLs containing the words and keep a tab on all the documents which contain the word. The frequency of the word in the HTML page i.e. the number of times the word occurs in the HTML page is also collected for the computation of BM25 Score. The position information of the words is not being stored as my program is not utilizing the position information for any boost to the scoring.

Thus the index files are being generated which basically serve as the infrastructure on which the search engine goes and hits to get the links pertaining to a particular search term or multiple search terms. There are three types of index files which are generated.

Following are the types of the index files generated and their significance in the term search:

1) Document index – This file contains all the URLs with corresponding program generated identifier which is used by the subsequent index files to identify the URL. The rationale for using document identifier (a number) is that a number can be compressed more efficiently than a string data. There is no compression done on this file
2) Lexicon index- This contains the terms and the corresponding metadata corresponding to the position of the data pertaining to the document identifiers containing the term which is stored in the inverted index file.
3) Inverted index file – This contains the document ids corresponding to the words in the lexicon index file. The metadata in the lexicon file refers to the position where the document Identifiers

pertaining to the search term has been stored, this helps in directly accessing the position and thus avoiding the situation of going through the complete file in order to read a single document identifier for a rare word in almost the end of the file.

The lexicon and document data explained above is then loaded into memory when the search engine is invoked and using the search terms, the inverted index data is accessed and the search results are prepared and output.
In case of multiple search terms being passed as search string, there are two types of search results which are generated. One is conjunctive search where the results URLs must contain all the search terms whereas in the disjunctive search the search results URLs should contain at least one of the search terms to be included in the final output.

The final output is of the form
<URL> <BM25 Score>

where BM25 Score can be represented as follows:

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

$$K = k_1 \times ((1 - b) + b \times \frac{|d|}{|d|_{avg}})$$

- **N: total number of documents in the collection;**
- **f_t: number of documents that contain term t;**
- **$f_{d,t}$: frequency of term t in document d;**
- **$|d|$: length of document d;**
- **$|d|_{avg}$: the average length of documents in the collection;**
- **k_1 and b: constants, usually $k_1 = 1.2$ and $b = 0.75$**

How to run the Program?

The code has been divided into four parts, the code needs to be run in the same order:

1) Build collections of all the words which are present in the HTML files and list them as and when they are seen. Here the entries are of the form
   <search term> <document identifier>
   <search term> <document identifier>
   ..
   Along with this the document index is also generated which contains the mapping between the URL and document identifier.

   The Code folder for this is "***WebSearchAssignment3BuildWordlists***"

To run the code, first create a file which contains just one number which will be used as the starting point for the generation of document identifiers. For explanation let us assume the name of the file is *documentIdSequence*.

Follow the following steps in the same order.

- Traverse to the main code folder WebSearchAssignment3BuildWordlists on a terminal window.
- Type the following command
  java –jar target/WebSearchAssignment3BuildWordlists-0.0.1-SNAPSHOT-jar-with-dependencies.jar <folder where the data dump containing the WET files are stored> <resultant folder where the index files will be generated> <absolute file path of documentIdSequence file>
- Check if files are generated, it will generate as many "wordlist_XX" files as the number of data dump GZ files are used to generate the postings collection.

Along with wordlist files, a document list file is also generated.

2) When all the files are generated, all the wordlist files need to be sorted and merged together to generate one file where all the lines are sorted on the basis of the term and document id
   Following commands are used to accomplish the above step:
   >for file in wordlist; do*
   sort -o $file $file
   done
   >sort -m wordlist>sorted_wordlist*
3) Using the sorted file generated, the lexicon file as well as the inverted index file is generated.
   The code folder "***WebSearchAssignment3MergeWordlist***" takes care of this.

   To generate the index files, follow the following steps:
   - Traverse to the main code folder WebSearchAssignment3MergeWordlist on a terminal window.
   - Type the following command
     java –jar target/WebSearchAssignment3MergeWordlist-0.0.1-SNAPSHOT-jar-with-dependencies.jar <Absolute path for the sorted word list file> <Absolute path for the file which needs to generated for inverted index> <Absolute path for the file which needs to generated for lexicon index>
   - To verify if it is working, check if the inverted index file and lexicon index files are generated.
4) Using the document index file, lexicon index file and the inverted index file, the search results are generated, disjunctive as well as conjunctive search results are generated. The code folder ***"WebSearchAssignment3SearchWords"*** takes care of this.

To perform the search, following steps to be followed:

- Transverse to the main code folder WebSearchAssignment3SearchWords on the terminal window.
- Type the following command:
  java -Xmx8g -jar target/WebSearchAssignment3SearchWords-0.0.1-SNAPSHOT-jar-with-dependencies.jar <Absolute path for the document index file> <Absolute path for the lexicon index file> <Absolute path for the inverted index file>
- When the prompt "Enter the terms you want to search (In case of multiple words, enter all the words separated by a space):" shows up, type the search query string and press Enter.
- The results that are output will have Disjunctive as well as conjunctive results in case of more than one search term in the search string. If only one word exists, then it will show a single search result which is a disjunctive one.
- The search terms in the Query string should be separated by space.

Framework used for developing the module

This module has been developed using JDK1.8. All the projects are Maven projects. Thus for running the module, JDK 1.8 will be a pre-requisite. For setup in addition to JDL, Maven also needs to be installed and added to the Path variable.

To compile/recompile any of the modules, follow the following steps:

1) Traverse to the main code folder on the terminal
2) Type "*mvn clean install*" and hit Enter.

How does it work internally?

For the complete working of the search engine, first of all the data crawled from the internet is to be loaded into the system so that this can be used as the source for word searches to be done. The data used in this search are the WET files. Following shows a sample WET file, please note this sample shows an excerpt for one URL; a WET file can have thousands of URL data within it along with the file header.

```
WARC/1.0
WARC-Type: conversion
WARC-Target-URI: http://news.bbc.co.uk/2/hi/africa/3414345.stm
WARC-Date: 2014-08-02T09:52:13Z
WARC-Record-ID:
WARC-Refers-To:
WARC-Block-Digest: sha1:JROHLCS5SKMBR6XY46WXREW7RXM64EJC
Content-Type: text/plain
Content-Length: 6724

BBC NEWS | Africa | Namibia braces for Nujoma exit
...
President Sam Nujoma works in very pleasant surroundings in the small but beautif
ul old State House...
```

WebSearchAssignment3BuildWordlists

To generate the word postings, the GZIP files from the source(http://commoncrawl.org/) have been first downloaded to the hard disk, this helps in accessing the files in the offline mode instead of directly accessing the Amazon S3 Servers. The data has been further cleansed to be able to get the dictionary words, numbers etc. without the symbols like full stop, hash symbols etc. Also the data from languages other than English have been filtered out to make the index more focused to English searches.

The words in the lines are split using the space symbol. This helps in getting all words present in any particular web page. Here we also record the total number of words present in the HTML page, this is a requisite data point for calculation of the BM25 score.

Every document/URL is mapped with a program generated sequence id so as to be able to run comparisons using integers instead of string URLs, also it is helpful in saving space in the inverted index. Thus in the end of this stage two types of files are generated; document index file (document ID and corresponding URL) and word posting files (list of word along with the document containing them.)

Unix Merge Sort

As the word postings generated are not in a sorted order and also are spread across multiple files, they need to be sorted on the word basis at least so that they can used to generate index files without much overhead.

Here all the word posting files are sorted and merged together to generate a consolidated sorted word postings file. The sorting is done on alphabetical order. We do not mind the document Ids getting alphabetically sorted as we take care of ordering them numerically in the next stage.

WebSearchAssignment3MergeWordlist

In this stage the sorted word posting data is utilized to generate lexicon index file as well as inverted index file. To reiterate the lexicon file contains the information about term, the number of documents the term is present it and pointer to the inverted index file where the list of the document ids containing the word are present. Also the inverted index contains metadata regarding document IDs blocks and the blocks themselves.

The sorted word posting file is read line by line, as soon as the value of the word changes, the data is written to the file to avoid memory overflow. The ordering of the document Identifiers is also taken care by the program here. To be able to compress the inverted list file, the document ids are compressed within a block. The metadata for the block like starting document id, number of values present in the compressed array are stored in the beginning of the block.

After all the blocks are written in the inverted list, the lexicon file is populated for the particular word.

To be able to compress the inverted index file, the following have been done:

1) Frequency starts from 0 instead of 1 i.e., every document's frequency is 1 greater than what it shows in the inverted index file. The frequencies have not been compressed though there is still an option to compress the frequency data as well.
2) For a block of document ids, every document id is subtracted with the first document id of the block thus reducing the size of the resultant value.
3) PFOR integer compression has been used which has been borrowed from a Java Port of a C implementation of PFOR integer compression. The credits for the compression jars can be found in the Reference section.

WebSearchAssignment3SearchWords

Now that we have the index files available with us, we use these files to search our database for the search strings.

In the beginning we load both the document index file as well as the lexicon file into the memory along with all the metadata contained in the document and lexicon files. To be able to cache the document Ids fetched for various search terms for future use, an LRU cache is also configured which deletes any older entry in case the number of entries is greater than the cache size.

Whenever a search string is input, the LRU cache is first hit to check if the document identifiers list for the word are also present or not. If the word is not found in the cache, then the lexicon data in the memory is queried to get the metadata for the word and get the position details in the inverted index.

After the document details are fetched from the inverted index, they are first decompressed and then mapped with their frequencies. Also the BM25 Score for the URL is also computed.

Now for the Conjunctive Search, all the common document IDs across all the words are fetched and the data is output in the descending order of the BM25 Scores. For the disjunctive search, the document IDs for all the documents are merged and the data is output in the descending order of BM25 Score.

For easy reference, only 100 links for Disjunctive Search are made available and 10 for the Conjunctive Search, this can be easily changed in the program to show all the entries if need be.

Below is a sample query run and the result for the same

```
[Pallabis-MacBook-Pro:WebSearchAssignment3SearchWords pallabichakraborty$ java -Xmx8g -jar target/WebSearchAssignment3SearchWords-0.0.1-SNAPSHOT-jar-with
-dependencies.jar /Users/pallabichakraborty/Desktop/WebSearchEngines/Homeworks/Homework3/index/doclist /Users/pallabichakraborty/Desktop/WebSearchEngine
s/Homeworks/Homework3/index/lexicon_list /Users/pallabichakraborty/Desktop/WebSearchEngines/Homeworks/Homework3/index/inverted_list
Loading the document and lexicon files
Document and lexicon files loaded
Enter the terms you want to search (In case of multiple words, enter all the words separated by a space):the music man
---------------------------------
Disjunctive Search Results
---------------------------------
http://www.audiosparx.com/sa/summary/play.cfm/crumb.1/crumc.0/sound_iid.262048 4.0956191732304985
https://www.sounddogs.com/extendedsearch.asp?Keyword=Viking 3.9495400218774916
http://www.audiosparx.com/sa/summary/play.cfm/crumb.2/crumc.0/sound_iid.315072 3.6705314304280248
https://www.gigaflat.com/browse.php?group=alt.binaries.e-books.german&order=date&sort=asc 3.564688590971907
http://www.getmein.com/tickets/man-of-moon-tickets/york-297768.html 3.5408071044775298
http://www.malianteo.com/foros/threads/programas.47916/ 3.421202149878692
http://starlifedancehall.com/starlife-dancehall/2014/2/22/mr-vegas-lashes-out-tackles-alkaline-matterhorn-in-new-single-newmusic 3.4179158956433677
http://www.make4fun.com/videos/Funny-man-videos/30681-A-little-don-juan 3.3674641647260755
http://flipmall.com/country.htm 3.361184186881566
http://gutenberg.net.au/ebooks09/0900821.txt 3.343969965425969
http://www.archive.org/stream/circuskingsourri00innort/circuskingsourri00innort_djvu.txt 3.3330580103101526
http://www.gamefaqs.com/gamecube/919043-mega-man-anniversary-collection/faqs/24150 3.324670641202366
http://www.2112.net/powerwindows/main/PushReferences.htm 3.3212388743292448
```

```
---------------------------------
Conjunctive Search Results
---------------------------------
http://starlifedancehall.com/starlife-dancehall/2014/2/22/mr-vegas-lashes-out-tackles-alkaline-matterhorn-in-new-single-newmusic 3.4179158956433677
http://flipmall.com/country.htm 3.361184186881566
http://gutenberg.net.au/ebooks09/0900821.txt 3.343969965425969
http://www.archive.org/stream/circuskingsourri00innort/circuskingsourri00innort_djvu.txt 3.3330580103101526
http://www.gamefaqs.com/gamecube/919043-mega-man-anniversary-collection/faqs/24150 3.324670641202366
http://www.2112.net/powerwindows/main/RushReferences.htm 3.3212388742202448
http://expectingrain.com/archives2001.shtml 3.3167276259672422
http://homepages.rpi.edu/%7Ebulloj/D-K.html 3.3153096289361157
http://www.sugarmtn.org/song.php?song=350 3.2988613656262173
http://www.archive.org/stream/germanachieve00cronrich/germanachieve00cronrich_djvu.txt 3.2972957371276306
https://placeit.net/stages/black-ipad-health-care-environment 3.296832300084724
Enter the terms you want to search (In case of multiple words, enter all the words separated by a space):
```

How long does it take on the provided data set?

Following is the timing for the processing of the files:

Processing done for 71 GZIP files.

1) Building word postings: 3 word posting files generated of cumulative size of 37.74 GB. This took around 2 hours to generate.
2) Sorting of the word lists and then merging them to one sorted file: Used Unix merge sort. This took around 12 hours to complete.
3) Building the lexicon and inverted index files from the sorted word list: 2 hours.
4) Searching Query String
   a) Loading of lexicon and document file to memory – Run once when the program is initiated – 2-3 minutes. During heavy load takes around 5-6 minutes.
   b) Searching data- Around 3-4 seconds, for very common words like “the”, “a” etc. it takes slightly longer.

How large the resulting index files are?

Following are the size details:

1) Document index file: 356.5 MB
2) Lexicon index file: 1.22 GB
3) Inverted Index file: 6.24 GB

What are the limitations it has?

Following are the limitations of the program:

1) The concept of chunks is not implemented thus the inverted lists do not get cached in the memory.
2) The scoring does not take into account the position of the words; thus conjunctive search will pick up any occurrence of all the words in the Query string

What are the major functions and modules?

The complete module has been broken into 3 major modules apart from the Unix merge sort operation.

The following are the major modules and the methods within the modules:

WebSearchAssignment3BuildWordlists

1) Class CreateProcessWordlists – This class contains the main method which calls the method *utilities.extractDataFromFolder* which as an output generates the word posting files along with the document index file.
2) Method utilities.extractDataFromFolder – This method takes the folder path where the zipped files of the crawled data is present. Along with this it also takes the absolute path where the intermediate word posting files are to be generated and also the absolute path for the file which contains the latest sequence number for the document identifier to be allocated to the URLs. In this method only the GZipped files are picked for processing. This method in turn calls the method *utilities.extractDataFromFiles* which extracts the data from the zipped files and generates the posting files.
3) Method utilities.extractDataFromFiles – This method takes as absolute path for zipped file, absolute file path for the word posting to be generated along with the document index file absolute path. It also accepts the sequence number containing file path.
   In this method for every URL the document identifier is set using the number available in the sequence number containing file, this number is then incremented and the file is updated for later use. The zipped file is read directly without explicitly unzipping the file.
   The file present in the zip file is read, the first 15 lines are ignored to ignore the file header for every file. Also the WARC header for every URL is ignored, however the URL is extracted from the tag WARC-Target-URI. Following is a sample WARC metadata available for every URL

   WARC/1.0
   WARC-Type: conversion
   WARC-Target-URI: http://news.bbc.co.uk/2/hi/africa/3414345.stm
   WARC-Date: 2014-08-02T09:52:13Z
   WARC-Record-ID:
   WARC-Refers-To:
   WARC-Block-Digest: sha1:JROHLCS5SKMBR6XY46WXREW7RXM64EJC
   Content-Type: text/plain
   Content-Length: 6724

   The data which is found in ASCII is only picked for the index creation, all the other data is discarded as they would not be of much use for the English word search functionality we are trying to implement. The words are split and then added in the word postings with their corresponding document identifier which indicate the URL in which they were present. In the same time the document index file is also written, as the document identifiers are being written and incremented, the document index file is automatically sorted when being written.

WebSearchAssignment3MergeWordlist

1) Class MergeWordlistsCreateIndices – This class contains the main method which is invoked to generate the lexicon index and inverted index files. The main calls the method *utilties.generateIndexFiles* which performs the task of generation of the index files.
2) Method utilties.generateIndexFiles – This method accepts as parameters the absolute path of the cumulative sorted word posting file, the absolute path of the inverted index file which will be generated along with the absolute path of the lexicon index file. The method reads the sorted file line by line, splits each line by the space obtaining the word and the document identifier. As the document identifiers are not sorted and can be duplicate thus they are put in a TreeSet. Also to keep track of the number of occurrence of the word in the document, the count of occurrence is also maintained in a HashMap along with the document identifier. Thus if there is a new word then the document identifier is inserted in both the TreeSet as well the HashMap. The count starts from 0 as we know that if the word has figured in the word posting then it has at least occurred once thus this can be adjusted during the word search. If a document Identifier already exists in the TreeSet then this means the it will be already present in the HashMap and the count of occurrence is incremented to take care of subsequent occurrences. In case the document count is equal to the pre-configured BLOCK SIZE then the documents are put in an array and compressed this largely reduces the size of the inverted index. Before compressing the value of the first document identifier in the block is subtracted from all the values to obtain smaller resultant values. After compression, the data is written along with the metadata into the inverted index block wise. This helps to prevent memory leakage or out of memory errors. After all the blocks for the word are written then an entry is inserted into the lexicon index file.

   Blocks are word specific in this implementation and do not spread across words. If there is a word for which there are not enough values, then only the whatever is present is written in the block.

WebSearchAssignment3SearchWords

1) Class SearchWords – This class contains the main method which takes 3 parameters, absolute path of the document index file, absolute path of the lexicon index file, absolute path of the inverted index file. Here when the main method is invoked, the document index file and the lexicon files are loaded into the memory first. A LRU cache is also configured which stores the already searched words along with their document identifiers so as to avoid an extra look up into the inverted index. LRU is configured using LinkedHashMap which gives an option to set a property where the oldest accessed value will be deleted in case the values in the data structure exceeds the pre-configured cache size.
   After which a prompt to enter the Query string is output in the command line. When the query string is received the program loops through every word to find the corresponding

document set. In case there is no result obtained for any of the words in the query string, a message stating that the unavailability of the word in the search engine is output.
For obtaining the document IDs for the words, the LRU Cache is first checked, if the word is found in the cache then the inverted index file is not accessed. If the LRU cache does not contain the word, then method *utilities*.*getDocumentIds* is called which fetches the document Ids for the word. The document Ids are added into the processing data structure as well as the LRU Cache.
After the document identifier details for all the words in the query string have been obtained, they are then forwarded for the disjunctive search method *utilities*.*calculateDisjunctiveSearch* as well as for conjunctive search method *utilities*.*calculateConjunctiveSearch*. If the query string comprises of only one word, then only a disjunctive search is executed.
For ease of evaluation, the output shown in the console contains 100 search results from the disjunctive search and 10 results from the conjunctive search. This can be easily modified in the code as per user requirements.

2) Method utilities.getDocumentIds – This method calls a method *utilities*.*readFromFile* which takes the details of the start position and end position of the entry for the word and then performs a random access on the inverted index file and reads the number of bytes specified. The data fetched contains all the blocks pertaining to the word. Every block is delimited by the separator "^". Within a block, the metadata of the block and the actual data is separated by "|" symbol to separate it from the actual block data which consists of compressed document identifiers followed by the frequencies.
   After the metadata is separated for a block and the first document identifier and the number of values in the compressed array is ascertained, the compressed array is first extracted, decompressed and all the decompressed values are added with the first document identifier value to get the original set of document identifiers.
   The document details are returned by the method along with computed BM25 Score. In the calculation of BM25, k1= 1.2 and b=0.75 are taken as default values.
3) Method utilities.calculateDisjunctiveSearch – This method is used to generate Disjunctive results for the search terms in the Query string. As all the document identifiers for all the search terms need to be added which can become a very huge list, thus we make use of TreeMap for the complete processing which is less memory intensive than data structures like HashSet. For all the document data which is received, a TreeMap data structure is populated where the document Identifier is the key, if the document identifier is found multiple times then the BM25 Scores are summed up and put into the Tree Map. To be able to output the data in the decreasing order of the BM25 Score, the foresaid TreeMap is used to populate another TreeMap which has the BM25 Score as the key and the values are a set of URL for the particular score.
4) Method utilities.calculateConjunctiveSearch – This method is used to generate Conjunctive results for the search terms in the Query string. Here as the document identifiers common to all the search terms need to be extracted, thus the usage of HashSet has been done, where for the first word's document set are inserted without any constraint after which the Set is adjusted to retain only the document identifiers which are common in the subsequent

word's document set, thus obtaining a set of only those document identifiers which are present for all the search terms in the Query String.
After this step, the BM25 Scores for all these document identifiers is summed up spanning through all the search terms and then this data is used to populate another TreeMap which has the BM25 Score as the key and the values are a set of URL for the particular score.

References

1) For the integer compression, Java port from C++ has been used, this is a part of
   Matteo Catena, Craig Macdonald, Iadh Ounis, On Inverted Index Compression for Search Engine Efficiency, Lecture Notes in Computer Science 8416 (ECIR 2014),
   2014. http://dx.doi.org/10.1007/978-3-319-06028-6_30
   Github Repository
   https://github.com/lemire/JavaFastPFOR
2) Tutorials and Presentations on using Common Crawl Data
   http://commoncrawl.org/the-data/tutorials/