PROFESSOR: MOHAMAD HODA
DUE DATE: FEBRUARY 26TH, 2018

INFORMATION RETRIEVAL AND THE INTERNET

ASSIGNMENT 1 REPORT / README

Leisha Danielle BURFORD
Student number: 7426797
Lburf102@uottawa.ca

Pauline ROMAGON
Student number: 300039464
proma070@uottawa.ca

INFORMATION RETRIEVAL AND THE INTERNET

ASSIGNMENT 1 REPORT / README

INSTRUCTIONS

To compile and run from the command line:

1. Add the files 'topics_MB1-49.txt', 'Trec_micorblog.txt', 'StopWords.txt' to the CSI4142_A1 directory (included for convenience)

2. Navigate to the included SourceCode directory

3. Compile with `javac -Xlint:unchecked Main.java`

Note: ignore the warnings

4. Run with `java Main`

Note: run takes ~4 minutes

5. Results for each step can then be found in the Results directory

6. In the event of unsuccessful compilation, the results that we achieved are in the Results_Done directory

DISTRIBUTION OF WORK

Pauline mainly worked on the *Prepro* and *RetrieveRank* classes and Leisha on the *Main* and the *Query* and *QueryParser*, and *InvertedIndex* classes. However, we both worked on each file to ensure overall consistency, and on the report.

CLASSES

PREPO

This class has 3 private attributes:

- stopwordsList (a list of strings, each string being a preprocessed stop-word)
- corpus (a list of list of strings, each list being a document, the first string of each list being the document ID and the others strings being the preprocessed words contained by the document, including duplicates),
- vocabulary (an ArrayList of strings, each string being a preprocessed word, different from stop-words; this list is free of duplicates).

The Prepro class has different methods:

- a *doPreprocess*(String stopWordsFile, String dataFile) method, that has to be called first (as it is the case in Main) in order to fill the attributes of the Prepro instance. It fills the corpus, the vocabulary and the stopWordList by reading the corresponding files and filtering the data. Among the preprocessing techniques, we did:
  - o Tokenizing
  - o Removing URLs
  - o Removing punctuation, and more generally non alphanumerical characters.
  - o Removing non alphabetical characters in tokens
  - o Stemming

  Once preprocessing is done and stopWordList is created, we fill the corpus and vocabulary attributes making sure stop-words are removed, and making sure there is no duplicate is the vocabulary.
- *getStopWordList*, *getCorpus*, and *getVocabulary* to get the corresponding attributes,
- *getDocLength* is a method particularly used in the RetrieveRank class in order to normalize the scores after having computed them; this methods returns an Hashmap<String,Integer> that maps each document ID to its corresponding document length.

STEMMER

This class is the Porter Stemmer class, called in the "Prepro" class to stem tokens. It contains multiple methods, however we mainly used the *add*, *stem* and *tostring* methods.

INVERTEDINDEX

This class creates the inverted index structure from a list of vocabulary words. The inverted index is implemented as a LinkedHashMap<String, HashMap<String, Integer>>. The LinkedHashMap was chosen over a regular HashMap to maintain the order in which the elements are inserted, which is sorted. Each word of the vocabulary is mapped to a HashMap, which maps docID to the term frequency for the word in this document. Note that the document frequency is not stored as it is just the size of the hash map of the associated word.

- The *addDocument*(String word, String documentID) method will add a document with the specified word to the postings list for this word. If this document is already in the list for this word then the termFrequency is updated otherwise, a new entry is made with termFrequency = 1
- The *getTermIDF*(String word) method calculates idf = log_2(# of docs / documentFreq) for the specified term The getTermDocumentWeight(String word, String documentID) method calculates and returns the w_ij = tf_ij * idf_i of the given word and document.
- The *fillNormalizedWeightIndex()* method computes the normalizing factors of each document by scanning the inverted index, then fills the NormalizedWeightIndex attribute by copying the inverted index and dividing the weights by the corresponding normalizing factors.
- The HashMap<String, Double> *getNormalizedWeight*(String word) method returns hashmap of documents with normalized weights for the specified word.

QUERY

This class is used to create instances of queries, it is called in the Main and the RetrieveRank classes. It has two attributes:

- id, a string containing the ID of the topic, like MB01.
- queryTerms, a list or strings where each string is a term of the query. This can contains duplicate terms.

This class contains usual methods such as methods to get and set each attribute, and also a method to print a query.

QUERYPARSER

This class aims at importing queries instances by processing the query file. The only method of this class, *getQueries*, takes as parameters the query-file name and a list of string of stop-words (returned by one of the Prepo methods); this method preprocesses the words of each query using the same preprocess that was used for creating the vocabulary, and then returns an ArrayList of instances of the class Query. Using the same preprocess allowed us to match words of the vocabulary and words of the queries.

RETRIEVERANK

This class has three attributes: an inverted index, a vector of document lengths (made from the same documents than the inverted index), and a list of queries.

It contains the following methods:

- *getQueryTermFrequencies*(Query query) returns a HashMap<String,Double> mapping each token of the query to its frequency in the query.
- *getNormalizedQueryWeights*(Query query) computes and normalizes the modified weights of each word in the query, by calling the frequencies retrieved by the getQueryTermFrequencies method. This methods then returns a HashMap<String,Double> mapping each token to its normalized weight.
- *getScore*(Query query), this function computes the cosine score for each document in the subset of documents containing at least one word of the query. It uses a term-at-a-time approach of computing scores. After having normalized scores, it returns a hashmap that maps documents ID to their normalized score, this hashmap is sorted by decreasing score value so that documents are ranked.
- *getRankedResults*, that calls the getScore method for each query, and returns a Map<Query, Map<String, Double>> mapping each query ID to its hashmap of result.

MAIN

This class articulates all classes and is the only file that needs to be run. The main method does the following:

- Instantiate a preprocess associated to the “Trec_microblog11.txt” and the “Stopwords.txt” files; then get the associated vocabulary, corpus and list of stop-words.
- Build the inverted index based on the vocabulary, and fill it based on the corpus.
- Parse the queries of the “Topics_MD1-49.txt”
- Get the result of each query by calling RetrieveRank’s methods, put the results in the right format and save them in a .trec file so that it can be evaluated.

Step 1: Preprocessing

We decided to represent stop-words and vocabulary words as Strings, and to store them in lists. We also decided to keep track of the words contained in each document by creating a corpus, which is a list of lists of strings, where the first string of each list is the document ID and the others are the preprocessed tokens; this corpus structure was mainly used to fill the inverted Index of step 2, in order to speed up the filling.

Step 2: Indexing

The hashmap structure chosen for building inverted index is a suitable structure for speeding up access to posting list; we organized it by word to make it match the example given in the assignment description, then the structure maps a word to its posting list (and the posting list is also a hashmap mapping a document ID to its frequency).

We also used the hashmap structure for another instance: the normalized-weight-index, it contains all the weights (tf-idf) normalized by document. The hashmap structures is not particularly suitable for the normalization part of this step, since it is organized by word and has to be normalized by document, however it has to be done only once and is useful when called by word for any query.

Step 3: Retrieval and Ranking

For this step, we chose a *term-at-a-time* score processing since the hashmap structures was suitable for retrieving weight by term.

Given a query, the scores to each document containing at least one word of the query were also stored in a hashmap mapping the document ID to the corresponding cosine, this structure was particularly helpful when computing the scores because it was storing the weights after each step of the term-at-a-time score processing. Once the scores computed, the pairs docID-scores were stored in a list so that it was sorted by decreasing scores.

Vocabulary

The vocabulary size is 53629.

The first 100 tokens are the following:

aa	aafaqu	aanzettten	abankinferi

aaa	aafia	aapkojashn	abarth
aaaaaaaaaaaaa	aag	aapl	abatesnttt
aaaaaaaaaaaaaaa	aaha	aargh	abba
aaaaaaa	aahaha	aarmaanta	abbevil
aaaaaaaaaaaaah	aahahah	aaron	abbeyniezgoda
aaaaaaaaargh	aahhahahahadio	aaronbertrand	abbi
aaaaaaawwwwww	aai	aarp	abbigliamento
aaaaaawwwwwwww	aaiisss	aarptx	abbotsford
aaaaah	aaj	aart	abbott
aaaaaiiiii	aalbamcfli	aartipaarti	abbrevi
aaaain	aambc	aashiyana	abc
aaaaronnnn	aamco	aaup	abcdefg
aaaggghhh	aamcocarcar	aawayi	abcenviron
aaah	aameen	ab	abcnew
aaahh	aamer	aba	abcnewsradio
aaand	aan	ababa	abcpolit
aaanniek	aanholt	abadi	abcrebirth
aaannnnddd	aanhoudingen	abajo	abcric
aaarghunknown	aankleden	aban	abcscienc
aaawwww	aankooppproc	abandon	abcthedrum
aaawwwwhhhh	aannnnyywhhere	abandonando	abcworldnew
aacon	aanstekelijk	abandono	abd
aacraoorg	aanunyu	abang	abdalla
aadithama	aanval	abank	abdallah
aadukalam			

We may think a majority of these words should have been removed from the vocabulary since no one of them are in an English dictionary, however they are actually useful for sentimental analysis tasks for instance.

Here are some other examples of the tokens that have particularly been stemmed and that are contained in the vocabulary: `appreci`, `beauti`, `bonu`, `calgari`, `cmon`, `favorit`, `fundament`…

Discussion of results

The top 10 result for queries 1 and 25 are the following:

MB001 Q0 33230810693763072 1 0.823 myRun1
MB001 Q0 29059262076420096 2 0.652 myRun1
MB001 Q0 34000879778533377 3 0.605 myRun1
MB001 Q0 34121208232415232 4 0.605 myRun1
MB001 Q0 33756527131107328 5 0.605 myRun1
MB001 Q0 29412843350654976 6 0.605 myRun1
MB001 Q0 33676211683069952 7 0.605 myRun1
MB001 Q0 28999698647883776 8 0.605 myRun1
MB001 Q0 34545260038193152 9 0.585 myRun1
MB001 Q0 34703780100448257 10 0.559 myRun1

Query 1 : BBC World Service staff cuts

1. Toyota Cuts U.S. Staff http://sockroll.com/ctgj2v
2. Is your staff safe? Who is training your staff? http://ow.ly/3FwQw

3. Cut it out
4. Cut her up
5. Cut
6. Cut.
7. cut
8. cut中止(笑)完全に暇になった
9. Photo: staff: http://tumblr.com/x021gadm93
10. Another great staff meeting at AHS. We have staff meetings every week in order to educate our staff & work out

MB025 Q0 32165530513178625 1 0.804 myRun25
MB025 Q0 31849003209461760 2 0.804 myRun25
MB025 Q0 31619184060272640 3 0.595 myRun25
MB025 Q0 31940780541083648 4 0.595 myRun25
MB025 Q0 29130585586794497 5 0.595 myRun25
MB025 Q0 32883405993545730 6 0.595 myRun25
MB025 Q0 31847865890377728 7 0.595 myRun25
MB025 Q0 29211688683180032 8 0.595 myRun25
MB025 Q0 32899672079208448 9 0.595 myRun25
MB025 Q0 32222922256941056 10 0.501 myRun25

Query 25: TSA airport screening

1. Screen.
2. Screen! http://plixi.com/p/73663737
3. @ airport
4. @ the airport already
5. at the airport :]
6. On my way to the airport(:
7. airport soon.
8. At the Airport..
9. Off to airport :)
10. Ah. get it off my screen.

Note: The tweet *"29735404542369793 New post: Seattle man acquitted in TSA airport case: See arrest video - Seattle Po http://bit.ly/e1W2Wo @TSABlogTeam #Big_Sis #Fascism #TSA"* would have been a good match too, but is too long to have the keywords weigh enough in its score.
In our results, the retrieved documents show that putting different weights to the query words could have impacted the result, for instance, adding a lower weight to the words "*screening*" and "*airport*" in the query would have be more consistent here.

Discussion of final results: The results are logical if we refer to the way they are computed: if we remove stopwords from the retrieved tweets of query 1 of query 25, they only contain "cut", "airport" or "screen" and then are good matches. However, they are not always relevant, even those with the highest scores. Thus, it would have been interesting to get a result of precision and recall of our Information retrieval system. An interesting approach could be to implement n-gram techniques so that a document containing a combination of words of the query would weigh more