Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Assignment 3

Information Retrieval

December 3, 2018

Professor Sérgio Matos

Diogo Ferreira ***76504***

Luís Leira ***76514***

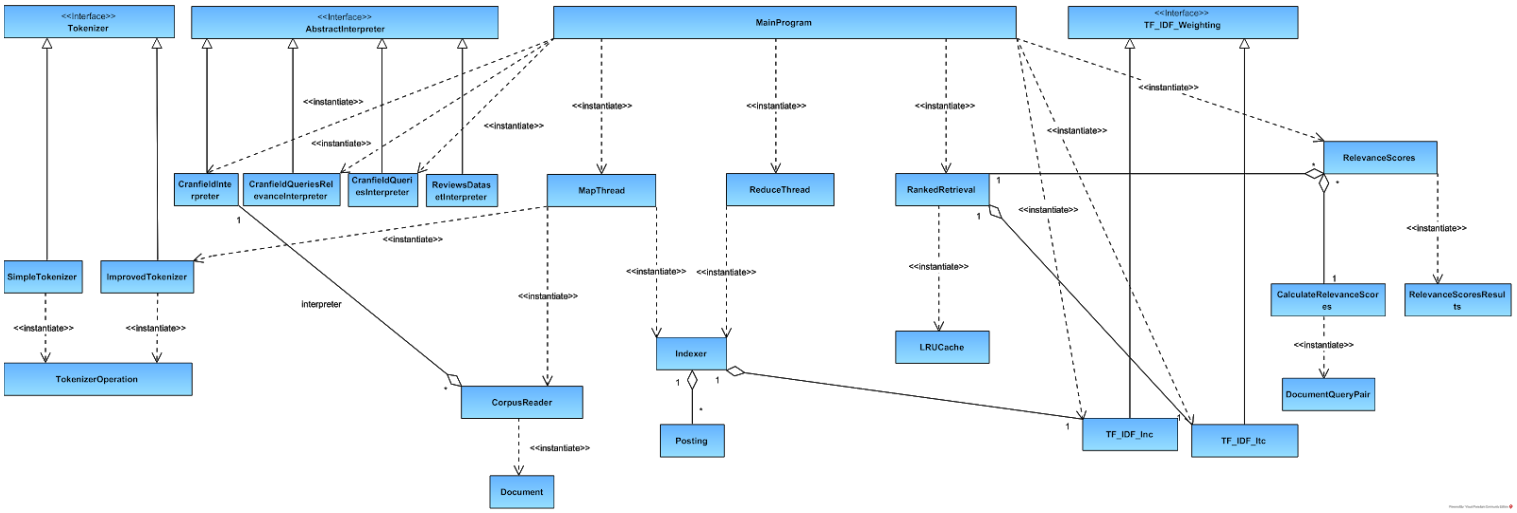
Context

This report explains the work done in the third assignment of the Information Retrieval course. This assignment's aim is to calculate performance metrics for our weighted retrieval method using the Cranfield corpus as the gold standard.

This work was made using the work previously done in assignment 1 and 2. The components unchanged from the previous version will also be mentioned, as well as the new or updated components, with the help of the architecture diagram. It also contains details about execution instructions, external libraries used, evaluation and efficiency metrics and final conclusions.

Architecture

In this section, the software architecture is presented in the form of a class diagram. Each class function and its main methods are described below. The new classes created in this assignment in comparison with the first and second assignment are: CranfieldInterpreter, CranfieldQueriesInterpreter, CranfieldQueriesRelevanceInterpreter, DocumentQueryPair, RelevanceScores, RelevanceScoresResults and CalculateRelevanceScores. The CorpusReader, MainProgram and Document classes were also updated.



MainProgram

This class contains the main flow of execution of the project. It has the responsibility of creating the workers for the map and the reduce phase (explained in the previous assignment 1). According to the maximum number of parallel threads and the maximum number of documents per indexer, the number of threads and documents per indexer will be adjusted.

After launching workers to execute the map phase and the reduce phase, the MainProgram will call finalizeIndex(int numberDocuments, String pathPrefix) to finalize the writing of metadata on the index files. Then, the exported index will be read from the files and imported into the application.

Three operations will be done with the indexer:

- The vocabulary size will be calculated;
- The first ten terms in only one document will be calculated with the method firstNTermsInMDocuments(Map<String, Integer> index, int n, int m);
- The ten terms with higher frequency in the indexer will be calculated with the method nTermsHigherDocumentFrequency(Map<String, Integer> index, int n).

After the processing of the index, a ranked retrieval instance is created, to search by queries in the indexed documents. The ranked retrieval instance will be used as an argument to the analyzeScores function, that given a ranked retrieval instance, a document with queries, its interpreter, a document with the relevance of the queries, its interpreter and the number of top elements to return, will return a RelevanceScoresResults instance, with the result of the evaluation metrics calculated with the gold standard.

MapThread

Each MapThread is responsible for reading part of the input file, specified by the start and end line. It starts by creating a corpus reader from a file and iterating through the allowed documents inside its range. For each document, it will apply a tokenizer and the resulting tokens added to the indexer. After indexing all the terms, the indexer will export the index to multiple files, each file containing the index with the terms that start with a different alphabetical letter. The index with all other terms that do not start with an alphabetical letter will also be saved into a different file.

ReduceThread

The ReduceThreads are launched after all MapThreads finish. Each thread is responsible for creating a unified index for the terms that start with a specific alphabetical letter (one thread is also responsible for creating the index with terms that do not start with an alphabetical letter). To do so, it is necessary to read all indexes with terms that start with an alphabetical letter and merge them all into one index. Because the indexes are sorted, it can be used a strategy to merge indexes without loading the entire indexes into memory. It can be specified a number of index entries to read from each file, and then sort them and append them into a temporary file. With this strategy, it is possible to construct large indexes without running out of memory. The long explanation of this algorithm is described in the Indexer subsection.

CorpusReader

The CorpusReader class deals with the opening, reading and pre-processing of the documents. It receives in the construction of a new object the path to the file where the corpus is, an interpreter to interpret each line as a document, a start line and an end line. The start line and the end line indicate a region from where to fetch documents from the file. The method *getNextDocument()* retrieves the next document object from the corpus in each call, and if there is no next document or the corpus reader has reached the end line, it returns null.

Document

The Document class represents a retrieved document from the corpus. It contains an ID (different for each document), the document title, the document authors, the document

bibliography and the document text. The Document constructor is private, and it is only possible to create a new document with the synchronized function createDocument. This Singleton construction causes the increment of the number of documents to be an atomic instruction.

AbstractInterpreter

The AbstractInterpreter interface contains the method *interpret(string doc)* and it is used for other classes that want to implement a method for retrieving documents from strings.

ReviewsDatasetInterpreter

The ReviewDatasetIntepreter implements the AbstractInterpreter and it is the implementation of an interpreter for the reviews dataset used in the previous assignments. It is currently not used in this assignment.

Tokenizer

The Tokenizer interface specifies the method *tokenize(String words)* that returns a list of tokens given the input. The input is a list of words to be tokenized.

SimpleTokenizer

The SimpleTokenizer class implements the Tokenizer interface to create a tokenizer. The *tokenize(String words)* method does four operations: splits the words on whitespaces, converts the tokens to lower case, removes all non-alphabetic characters and removes terms with less than three characters.

ImprovedTokenizer

The ImprovedTokenizer class is another implementation of the Tokenizer interface. The operations available are the following ones:

- splits the words on whitespaces;
- converts the tokens to lower case;
- removes invisible characters;
- removes all tokens that are e-mails;
- removes all tokens that start by a number and have a punctuation next to the number;
- removes meaningless characters;
- removes tokens that are hyperlinks;

- removes full stops at the beginning or end of token;
- removes all “^” characters without numbers on both sides of it (can indicate an exponential number);
- removes hyphenation at the beginning and end of tokens;
- removes the stop words according to a list of stop words;
- stems the tokens with the Porter Stemmer in English language.

  The operations are carefully described in the TokenizerOperations class subsection.

TokenizerOperation

The TokenizerOperation class has the methods to apply transformations to words and turn them into tokens. When the class is loaded, the stop words are loaded from a file, using the method loadStopWords(). When the class is instantiated, the constructor receives a string of words that transforms into a list of tokens, splitting the words on whitespace. The process results on a stream of tokens, that is stored inside the object. The stream of tokens can be retrieved using the method getTokens(). The operations can be piped into each other because the return type is the TokenizerOperation class. The methods that can be applied to tokenizer the words are:

- removeNonAlphabeticCharacters() – Removes all the non-alphabetic characters from the tokens;
- removeTermsWithLessThanNCharacters(int n) – Removes all tokens with less than n characters;
- toLowerCase() – Converts all token characters to lower case;
- stem() – Using the Porter Stemmer, applies the stem operation to all tokens;
- removeStopWords() – According to the loaded list of stop words, removes all the tokens that are present in that list;
- removesNumbersFollowedByPunctuation() – Removes all the tokens that start by a number and have punctuation next to that number (common problem in one analyzed corpus);
- removeMeaninglessCharacters() – Removes list of predefined characters from tokens. The list of characters is composed by special characters, like ‘$’, ‘!’, ‘[’ or ‘?’;
- removeSoloHyphenation() – Removes the hyphens from tokens where they are in the beginning or end of the token. Remove also all consecutive hyphens;
- removeHyperlinks() – Removes all tokens that are hyperlinks;
- removeFullStops() – Removes full stops at the beginning or end of token;
- removeExpoentSymbolWithoutNumbers() – Removes all ‘^’ characters that do not have a number by each side. When there is a number on both sides, the character is kept, because it may symbolize an exponential operation;

- removeEmails() – Removes all tokens that are e-mails;
- removeInvisibleCharacters() - Removes invisible characters that can be fetched when reading from a file.

Indexer

The Indexer class has the goal of creating an inverted index composed by the terms and its postings. The indexer will consist in a TreeMap, with the terms being the keys and a TreeSet of Postings as the values. The TreeMap and TreeSet both have complexity O(log2(n)) for insertion and search and keep terms and postings ordered.

The addToIndexer(List<String> tokens, int docID, TF_IDF_Weighting weighting) method receives as argument a list of tokens, the id of the document where the tokens were retrieved and the weighting scheme, and it will add the terms to the indexer along with its weights and its positions inside each document, using the method addTermToIndexer(String term, int docID, double termWeight, List<Integer> positionsInDocument) to each term.

The method addTermToIndexer(String term, int docID, double termWeight, List<Integer> positionsInDocument) starts by checking if the term is already on the indexer. If it is not found, it creates a new entry in the index and associates with it a new TreeSet with one posting. It the term is found, retrieves the list of postings of the term and the new posting is inserted into the list.

The function Map<String, TermValue> calculateTermsWeight(Map<String, List<Integer>> termsInDocument, TF_IDF_Weighting weighting) receives as argument a list of terms in the document and its positions in the document, as well as a weighting scheme to weight the documents. This function will calculate the term frequency weight, will apply normalization, and it will return a map with each token and a TermValue class, that will contain two records: the term weight and the positions in a document of each term.

The exportToFileByLetter(String prefix) method exports the created index to files. The index will be separated into multiple sub-indexes. 27 files can be generated, each one with the content of each sub-index. Each sub-index is identified by the first letter of its terms (26 indexes for a to z, and one index for other characters).

The mergeMultipleIndexes(String path, String postfix, String outputFilePath, int maxDocumentsInIndex) method merges the indexes created in the exportToFileByLetter(String prefix) by the first character of the terms in one index. To do so, receives as argument a postfix that all index files follow, to list them. Then, the reading of the indexes in those files is made in sequence. For all files, it will be read a limited number of lines, to not cause an out of memory error. Because the indexes are already sorted, the read index entries can be merged and sorted easily. It is kept track of the last term read in each file. When the number of lines is read from all files, it is searched for the last read term in all files with a lower value. It can be proved that from the beginning of the index until that term all entries are sorted, and can be appended into a file. The entries written into the file are removed from the index in memory and the files are read again. This process continues until all files are fully read. In the end, a complete index is outputted to a file, resulting in the merging of multiple indexes.

The finalizeIndex(int numberDocuments, String pathPrefix) method is to be called when the index files are all written, and it writes the index metadata in a separate file, containing the number of analyzed documents and the number of terms in the indexer.

Posting

The Posting class contains a document ID, the weight of the term in the document and a list of positions in the document where the term can be found. This class also implements the Comparable interface, to compare two postings by the document ID. This feature is useful to automatically sort the Posting instances in a sorted list, in the indexer.

TF_IDF_Weighting

This interface has the methods needed to implement tf-idf weighting: calculate the term frequency, calculate the inverse document frequency and normalization of terms.

TF_IDF_lnc

This class implements the IF_IDF_Weighting interface using lnc weighting scheme: log-weighted term frequency, no IDF and cosine normalization.

TF_IDF_ltc

This class implements the IF_IDF_Weighting interface using ltc weighted scheme: log-weighted term frequency, IDF weighting and cosine normalization.

RankedRetrieval

This class is used to search by a query in the indexed documents and to return the most relevant document IDs. The only function exposed for public access is the search(String query, int numberDocumentsTop, boolean proximitySearch) method, that given a query and the maximum number of documents to retrieve, returns a list with the ID's of those documents. The argument proximitySearch allows the caller to enable proximity search, only returning documents that have queries that exactly match the query made.

In the instantiation of an object of this class, it is required to pass as argument a tokenizer, to tokenize the queries made, a TF-IDF weighting scheme, the folder name where the index is, the maximum size of the posting lists in memory and a ratio to append the terms into cache. The maximum size of the posting lists will be used to limit the size of the cache to retrieved search documents. The maximum ratio to append terms in cache is a ratio that refers to the possibility of caching all found terms and posting lists, or cache only the searched ones. If the maximum number of posting lists in memory divided by the number of terms in the indexer in bigger than this ratio (the memory size is large relatively to the number of terms in the indexer), all terms found will be appended in the cache. Otherwise, only the searched terms will be appended. Internally, the Ranked Retrieval will get from the

index metadata the number of documents in the index, to be used in the term weighting phase, and the number of terms in the indexer, to calculate the ratio to append all terms into cache.

Some relevant private functions are:

- Map<String, Double> calculateQueryTermsWeight(List<String> tokensList) - GIven a token list of a query, calculate the weight of the terms (term frequency, IDF and normalization);
- Map<Integer, Double> cosineScore(Map<String, Double> queryTermsWeight, int numberDocumentsTop) - Calculates the cosine between a query and all documents in the indexer, and returns a list with numberDocumentsTop documents ordered by its cosine score. The argument queryTermsWeight contains a map with the terms in a query and its respective weight.
- Map<Integer, Double> cosineScore(Map<String, Double> queryTermsWeight, int numberDocumentsTop, List<String> queryTokens) - Calculates the cosine between a query and all documents in the indexer that exactly match the query with the tokens in the same sequence, and returns a list with numberDocumentsTop documents ordered by its cosine score.

  Due to its complexity, this function deserves further explanation. For the analysis of the first token in the query, the cosine score will work just the same as the non-positional cosine score: for each document in the posting list of the token, it will add to the score of that document its term weight multiplied by the term weight in the query. It will also store the positions of the term in the document.

  For the next tokens in the query, the function will go through all the documents that have already been processed (i.e. documents where all the analyzed tokens are in the same sequence in the document as in the query) and it will check if they are in the posting list of the term. If they aren't, they are removed from the processed documents, and its score for the query is 0. If they are in the posting list, it will check if any position of the analyzed token in the document is next to any position of the last token in the same document. If it isn't, the document is removed from the processed document and its score in the query is 0. If it is, updates the last positions of the token in the document in an auxiliary structure and it will check if the token weight was already added to the document score. If not, updates the document score with its term weight multiplied by the term weight in the query.
- Map<String, Integer> calculateDocumentFrequency(Set<String> terms) - Given a set of terms, calculate the number of documents that each term has in its posting list.
- int getNumberOfDocumentsInIndexer(String pathPrefix) - Get the total number of documents in the indexer.
- List<Posting> getPostingList(String term) - Given a term, returns its list of postings, or returns null if the term does not exist in the index. To speed up the search phase, relies on an LRU cache that stores the recently accessed terms and its posting list. The LRU cache removes the eldest entry when maximum size of the cache is reached. If the number of indexed terms is not too big when compared to the size of the cache, all terms and its posting lists will be cached when found in the indexer. In this way, it is possible to cache most terms before even searching for them. Otherwise, if the number of terms is too large relatively to the available memory, only the searched terms will be added to the indexer. In this way, our goal is to optimize

the cache utilization, reducing the cache misses. The searched terms not found in the index are also cached with null value, to prevent from accessing the file when searching for them.

LRUCache

Implementation of a Cache with the replacement policy of Least Recently Used. It extends the implementation of the LinkedHashMap and overrides the removeEldestEntry to replace the eldest term when the maximum size of the cache is reached.

RelevanceScores

This class has only one static method, analyzeScores(RankedRetrieval searcher, String queryFilename, AbstractInterpreter queryInterpreter, String relevanceFilename, AbstractInterpreter queryRelevanceInterpreter, boolean proximitySearch, int numberOfResultsToRetrieve), that analyzes the relevance score of a list of queries given a search engine and a gold standard. This method receives the searcher, the filename where the queries are, the interpreter to that file, the filename where the relevance of the queries are, the interpreter to that file, a proximity search flag to indicate if the search is to be done with proximity search and the number of top documents to retrieve.

The method analyzeScores will start by storing the results of all the searches done for the queries in the file in a map. Simultaneously, it also measures efficiency metrics: the query throughput and the median query latency. After the search results for all queries are stored, it will load all the document IDs and its relevance for each query. Each pair DocumentID-Query will be stored on an instance of the DocumentQueryPair class, and the relevance for each DocumentQueryPair will be stored in a map.

Finally, both maps will be send as arguments to the various functions in the CalculateRelevanceScores class and the resulting metrics will be saved in an instance of the RelevanceScoresResults class, that will be returned.

CalculateRelevanceScores

This class implements the methods needed to calculate the relevance scores of an index. All methods receive two arguments: a map with the results of the search of each query and a map with the relevance for each document for a query, with exception to the method calculateMeanPrecisionRankN, that also receives N as a parameter (explained below). All functions return a double number, that represents the score between 0 and 1. The implemented functions are:

- calculatePrecision - For each query, divide the number of retrieved documents that are relevant by the total number of retrieved documents, and average all results.
- calculateRecall - For each query, divide the total number of retrieved documents that are relevant by the total number of relevant documents, and average all results.

- calculateFMeasure - Calculate the measure of a test's accuracy, multiplying the precision times the recall times two, and dividing by the sum of precision and recall.
- calculateMeanAveragePrecision - For each query, calculate the average precision, given the position of the retrieved document in the search, and average the results for all queries.
- calculateMeanPrecisionRankN - For each query, calculate the precision only having into account the first N documents (N is passed as a parameter). At the end, return the average results for all queries.
- calculateNormalizedDiscountedCumulativeGain - For each query, first computes the actual discounted cumulative gain. Then, sorts the relevant documents by its relevance and computes the ideal discounted cumulative gain. Finally, for each query, divide the actual discounted cumulative gain by the ideal cumulative gain. At the end, return the average across all queries.

RelevanceScoresResults

This class stores the relevance scores results. The stored metrics are:

- Precision
- Recall
- F-measure
- Mean Average Precision
- Mean Precision Rank N
- Normalized Discounted Cumulative Gain

DocumentQueryPair

This class represents a pair Query-Document ID and it is used as key in the map with the relevance of the documents in the queries in the gold standard.

CranfieldInterpreter

This interpreter implements the AbstractInterpreter interface and interprets the documents in the Cranfield corpus, returning the document number, the title, the authors, the bibliography and the test.

CranfieldQueriesInterpreter

This interpreter implements the AbstractInterpreter interface and parses the queries for the Cranfield corpus for evaluation purposes.

CranfieldQueriesRelevanceInterpreter

This interpreter implements the AbstractInterpreter interface and parses the file with the documents and the relevance in a query. Returns a list with the query ID, the document ID and the relevance of each document for a query.

External libraries

The different tokenizer implementations have two methods that use an external library whose documentation can be seen in http://snowball.tartarus.org/. The stem method is an integration of the Porter stemmer according to the library, whose information is available at http://snowball.tartarus.org/download.html. The removeStopWords method is a stopword filter that uses a default list according to the same library, whose information is available at http://snowball.tartarus.org/algorithms/english/stop.txt.

Execution instructions

There are two main ways to compile and execute the code: by the command line or with an IDE. Both are explained below.

The parameters available to change (*MAX_PARALLEL_THREADS*, *MAX_DOCUMENTS_PER_INDEXER* and *MAX_POSTING_LISTS_IN_MEMORY*, explained in assignment 2 report) are in the MainProgram. The file path of the corpus and the tokenizer are also specified in the main program.

Command line

To use this method, Maven must be installed.

Go to the project root directory.

The external library must be integrated by doing the download of the libstemmer (available at http://snowball.tartarus.org/download.html) and following the README instructions.

Then, build the project by executing the following command:

```
mvn package
```

After building the project, you may test the compiled and packaged JAR with the *MainProgram* from the solution by executing the following command:

```
java -cp target/IRProject-1.0-SNAPSHOT.JAR assignment3.MainProgram
```

IDE

This project can also be imported in any IDE that supports Maven like NetBeans or Eclipse.

After importing the project, do “Build” and then go to Run > Configurations > Customize > Run and select the *MainProgram* from the solution. No arguments are needed to run the application.

Finally, click on “Run project”.

Evaluation and Efficiency Metrics

On this section, we will provide the results of the evaluation and efficiency metrics of the searches done on the “Cranfield” corpus.

Evaluation Metrics

Using the SimpleTokenizer and retrieving 100 documents from the searches, our results were the following:

- Precision: 4,27%
- Recall: 64,53%
- F-Measure: 0.0801
- Mean Average Precision: 25,39%
- Mean Precision Rank 10: 16,53%
- Normalized Discounted Cumulative Gain: 0.3230

Using the ImprovedTokenizer and retrieving 100 documents from the searches, our results were the following:

- Precision: 4,55%
- Recall: 70,34%
- F-Measure: 0.0855
- Mean Average Precision: 26,71%
- Mean Precision Rank 10: 17,06%
- Normalized Discounted Cumulative Gain: 0.3519

Efficiency Metrics

These efficiency metrics were taken in a 3 year old laptop with an *Intel i7-4720HQ (4 cores, 8 threads) 2.60GHz*, *16GB RAM, 1TB HDD disk (5400rpm)* running *Windows 10*. The results presented are the average of 5 running times.

Using the SimpleTokenizer and retrieving 100 documents from the searches, our results were the following:

- Query throughput: 569.6116 queries/second
- Median query latency: 0.6853 ms

Using the ImprovedTokenizer and retrieving 100 documents from the searches, our results were the following:

- Query throughput: 636.1914 queries/second
- Median query latency: 0.8260 ms

Conclusions

On this assignment it were implemented several evaluation metrics over a gold standard corpus: Precision, Recall, F-measure, Mean Average Precision, Mean Precision at Rank 10 and Normalized Discounted Cumulative Gain (NDCG). The query throughput and the median query latency were also measured as an efficiency metric.

Our results show that the implemented searcher achieved relatively low precision metric. That fact may happen due to the high number of documents retrieved (100). If the number of returned documents in the search is lower, the precision will increase. However, it will prejudice the recall. It can also be seen that the personalized tokenizer (ImprovedTokenizer) slightly improves the overall results.

In the efficiency results it can be seen the high query throughput and the low median query latency achieved. These efficiency metrics are determinant when the model is deployed in large scale, something not thoroughly tested in this work. However, it proves correct the claims about the scalability, flexibility and efficiency of the LRU cache made in the second assignment report.