Project 5: Information Retrieval: A Memory-based Search Engine

Contents

1 Overview

Please refer to the lecture slides for the foundations of Information Retrieval (tokenization, inverted indices, efficient boolean search and document scoring to generate ranked query results), which are assumed prior knowledge before reading this document.

1.1 Information Retrieval

The overall project is to build an in-memory information retrieval system using a HashMap-based inverted index and the TF-IDF scoring rule from information retrieval.

You are recommended to begin by looking at test.TestSearchGUI to understand the basic search framework. (You do not need to understand test.TestSearchGUI other than modifications to the main method useful for testing your code.) After this, look at test.TestSearch to see how the document indexing and search are called externally and to get a sense for the pieces you have to implement in the tasks below and how they all support the search results.

1.2 Tasks

Your tasks are to create and complete the following classes and thoroughly test your code to ensure it matches the solution as demonstrated in test.TestSearch. **Ensure that you use the package and class names specifed; failure to do this will prevent autograding and lead to a 0.**

We note the handout code is only lightly commented; most functionality should be evident from the class and method names as well as the parameters passed to methods.

- package io: class FileDocSource extends DocSource

  A DocSource simply provides an interface to a set of documents that the search engine can refer to by document ID and access to build the search index. StaticDocSource is given as an example.

  FileDocSource should **load all files from a directory name provided in the constructor as a** String. The class FileFinder handles recursively finding all files and should be used in the solution.

  FileDocSource **should not load all files into memory at the same time**; it simply stores an array of all filenames to be loaded on demand as required by method calls to the class.

  Note that the "document ID" is just the array index of the filename. By using methods in FileFinder to obtain a list of files under a directory, you will guarantee the same array order and index values as the solution.

  We provide you with a **zip file of NSF research grant abstracts** from the early 1990's for testing your code on a fairly large corpus of documents – please see Blackboard for this zip file which you should

uncompress. **Do not store these data files in github; github is primarily for code and the autograder will time-out when attempting to checkout your repository for grading.**

- package tokenizer: class IndexingTokenizer implements Tokenizer

  In the indexing pipeline of information retrieval, after a document is loaded, it is tokenized.

  SimpleTokenizer is given as an example of how a document can be tokenized into individual terms as the first stage of indexing.

  IndexingTokenizer should modify SimpleTokenizer in two ways: (1) it lowercases all tokens and (2) it preserves hyphenated words as single words, i.e., "state-of-the-art" should be a single token.

- package index: class SortedDocScore extends DocScore implements Comparable

  This is a simple helper class that represents a scored document resulting from a search query. If you add all SortedDocScore's to a TreeSet, the TreeSet will allow you to iterate through all results in sorted (i.e., rank) order.

  SortedDocScore's should be sorted first by score (highest score first) and second alphabetically by content. If score and content match, they are considered equal.

- package index: class InvertedIndex extends Index

  With all of the above ingredients, it is now time to complete the main class and algorithms for this project.

  You must implement an inverted index data structure that is a concrete subclass of abstract class Index (i.e., you have to implement all of its abstract methods). This class indexes all of the documents in the DocSource and provides an interface for efficiently searching them and returning ranked results scored by TF-IDF once the index is built.

  The InvertedIndex structure should be stored in memory in a private

  ```
  HashMap<String,HashMap<Integer,Integer>> index;
  ```

  that maps terms (i.e., tokens from your Tokenizer) to a nested HashMap of { doc ID → term frequency in that doc ID }. This is the only configuration of the inverted index that will receive full credit.

  The document frequency for each term (needed by the TF-IDF calculation) should also be computed at indexing time and be stored separately in private HashMap<String,Integer> _docFreq;

  that maps terms to their document frequency. Note that document frequency is not collection frequency – it is the count of documents that contain the word and is capped by the total number of documents; many students will incorrectly count the overall frequency of the term in the collection, which you should observe is not the correct definition of document frequency.

  The two key methods in InvertedIndex are buildIndex and getSortedSearchResults; the former builds the inverted index as described above from all files in the DocSource, while the latter supports efficient search and ranked information retrieval as described next.

  The implementation of getSortedSearchResults should make heavy use of the Collections classes and is **not meant to follow the merging approach of the lecture slides**. The lecture slides are intended for disk-based indexing and retrieval while in this project we are building an in-memory index. For this reason, **as you process each tokenized query term**, you should simply maintain a HashMap from document IDs to a document's current score (if non-zero); as each query term is processed, the score for that term in a document can be added to that document's current score in the HashMap (note that term scores are provided by InvertedIndex's scoring data member). Once all query terms are processed, you should create SortedDocScore's for all matching documents and their non-zero scores. getSortedSearchResults should return a list of SortedDocScore's sorted by the ordering defined above for SortedDocScore.

- package score: class TFIDFScoringFun implements TermScoringFun

TFScoringFun is given as an example of the simple TF scoring rule for terms. You should use TFScoringFun when initially writing the InvertedIndex above. Then you can move on to the TFIDFScoringFun next.

The TFIDFScoringFun uses the TF-IDF term scoring rule from the lecture slides where $TFIDF_{d,t} = \log_{10}(1 + TF_{d,t}) \cdot \log_{10}(\frac{N}{DF_t})$ (use Java's Math.log10() for this). Here $TF_{d,t}$ is the frequency (count) of term *t* in document *d*, DF_t is the frequency (count) of the number of documents in the corpus (data set) that contain term *t* and *N* is the total number of documents. See the lecture slides for a justification of this term scoring function.

Observe that the use of DF_t and *N* require access to the search Index, since it contains methods that provide access to the document frequency of all terms in the corpus as well as the total count of all documents in the corpus.

All terms are assumed to have minimum document frequency 1, so if a term does not appear in the document corpus, its document frequency should be assumed to be 1.

Note that this class only scores a term; the search algorithm in the InvertedIndex will take care of summing these term scores over all matching query terms in the document.

- JUnit: **You are required to write five** JUnit **test cases that are submitted with your assignment**; these will not be autograded (so their exact location in your repository is your choice) but they will be examined during the code review. Keep in mind that plagiarism will be checked for these JUnit test cases. You should be able to explain the rationale behind your five test cases and you should test individual class and method functionality as well as overall search results that test the entire project end-to-end.

2 Helpful Hints

- We will not grade the output of toString() methods or other text output to the console.
- Because all of your classes and the solution classes implement the same subclasses/interfaces, it is easy to interchange one class at a time as you implement it to ensure it produces the same output as the solution. However, be careful that you do implement and test all of your classes (we will test them); your final solution should not require any classes from the soln package.
- To load the unzipped file of NSF abstracts efficiently, you may need to **increase memory** to the Java JVM by setting the flag -Xmx1000m which allocates up to 1000Mb to the JVM. Netbeans allows you to set this flag in "VM Options" accessible from the project properties view; Eclipse allows you to set this from the Run→Run Configurations, Arguments tab using the "VM arguments" box. Google if you need more help increasing the so-called "heap size" in your IDE.
- Please do not change the package or filenames from what is specified in this document and the starter code or your project cannot be autograded.
- Please do not commit directories of the text data files to github... the autograder will not be able to checkout your project and you will get a 0. The easiest way to avoid this mistake is to place the files **outside of your repository directory**.
- Test your code thoroughly with test cases of your own design. Consider multiword queries and fringe cases such as queries containing words that are not in the test data collection.
- Refresh yourself on the Collections class hierarchy (subclasses of Collection, SortedSet, etc.), the methods available to each class, how to specify the sort ordering for sorted Collections, etc. Much of this material was covered in lecture and in Java examples provided on Blackboard, but Google will also provide excellent resources for augmenting the lecture content.

- Your code should not run substantially slower than the solution.
- If you need to do a late submission, be sure to `git pull` first to update your repository with the autograde file from the first submission. If an editor pops up, just type `:q` to quit (perhaps enter as well). After you've pulled, you should be able to commit and push your code.
- **Please remember that plagiarism is a serious academic offense.**