COMP 345 Advanced Program Design with C++

Fall 2017

Assignment #2

Due date (Moodle Submission): October 17
Due date (Lab Demo): October 18 & November 25

Task 2: "Search Engine" (8%). Develop a C++ program that provides a full-text document query interface, similar to a search engine like *Google*. Your program has to:

1. Index a set of text files, provided in the same form as in Assignment #1.
2. Then take *queries* from the user (i.e., a list of keywords), and for each query output the top-n files that best match the query, together with their *score* as defined below (n is a parameter you can set in your program, e.g., $n = 10$ to display the top-10 files matching the query).

Document indexing. The search index you have to construct is an enhanced version of the document-term matrix from Task 1 (Assignment #1): Instead of only computing the simple word count for each document, now compute the following values:

- Document count: N is defined as the total number of documents in your index
- Term frequency: we will now call the word count from Task 1 the *term frequency*, denoted by $\mathrm{tf}_{t,d}$, where t is a term (token) in your dictionary and d is a document you indexed[1]
- Document frequency: the *document frequency* df_t for a term t is defined as the number of documents that t appears in[2]
- Tf-idf weight: the *tf-idf weight* of a term t in a document d is defined as:

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

You can now represent each document d as a *vector* $\vec{d}$ of tf-idf weights (this is a *vector* in the mathematical sense—it does not have to be implemented with the STL `vector` class!).

Processing the query. Convert the query into the same vector representation as the documents. The result is a query vector $\vec{q}$. Tokens from the input query that do not appear in the indexed documents are simply ignored.

Document ranking. Now we have to find the documents that best match the query. This is done using a scoring function as follows: For each document d_j in your index, compute the *cosine similarity* between the document vector $\vec{d_j}$ and the query vector $\vec{q}$ as

$$\mathrm{score}(\vec{q}, \vec{d_j}) = \frac{\vec{q} \cdot \vec{d_j}}{||\vec{q}|| \cdot ||\vec{d_j}||} = \frac{\sum_{i=1}^{N} w_{i,q} \cdot w_{i,j}}{\sqrt{\sum_{i=1}^{N} w_{i,q}^2} \cdot \sqrt{\sum_{i=1}^{N} w_{i,j}^2}}$$

with $\vec{q} = (w_{1,q}, \ldots, w_{N,q})$ and $\vec{d_j} = (w_{1,j}, \ldots, w_{N,j})$. You will receive a value in $[0, 1]$, where 0 is the lowest possible score (no tokens in common) and 1 the highest. You can now create the rank of the top-n documents for this query, sorted by score (highest on top), and present it to the user (displaying the file name you read in, together with the score, is sufficient). Ask for the next query, until the user terminates the program (i.e., with end-of-file).

---

[1] That is, $\mathrm{tf}_{t,d}$ is one entry in your old document-term matrix – in the table on the Assignment #1 sheet, $\mathrm{tf}_{java,d_3} = 3$

[2] So, if you index 10 documents, and "hello" appears in 3 of them, then $\mathrm{df}_{hello} = 3$

Congratulations! You just implemented a basic form of the *vector space model* used in *Information Retrieval (IR)*. To compete with Google, now you just need to add a Web interface and index all documents on the Internet (no additional marks, sorry).

Coding guidelines. Develop your program according to the following specification:

a) For all classes, make sure you properly separate your system into *header* (`.h`) and *implementation* (`.cpp`) files. Put each class into its own translation unit. You are free in the choice of an IDE, but your code must be standard, cross-platform C++ code.

b) Document all your classes and functions with *Doxygen*.

c) For your classes, follow object-oriented design principles as discussed in the course; in particular make data members `private` unless you have a good reason not to; and use `friend` functions where appropriate to access private members.

d) Write two separate `main` programs, using the *same* classes (see *e)* below): a new `indexer.cpp` that implements Task 1 from Assignment 1 (printing the document-term matrix, but now also print the added document frequency and tf-idf weights); and a program `googler.cpp` for the new search Task 2 (you will have to demo both main programs).

e) Design your new code around the following classes and methods:

Class `document`: A document object `d` represents one document in your index. It provides a default constructor, which creates an empty document, as well as a constructor that accepts a file name and reads the file contents into the document object. Each document provides the functions `name()` (returns the file name of the document), `size()` (size in characters), and `content()` (returns the text of the document).

Class `stopword`: An object of this class can be used for stopword filtering. The default constructor creates an object with an empty stopword list (i.e., no filtering). A constructor with a file name reads the stopword list from this file. To check if a given token exists in a stopword object, overload `operator()` to return `true` if the token is in the object's list, otherwise `false`.[3]

Class `tokenizer`: This class is responsible for breaking an input stream of characters into individual tokens, by splitting it at white space and punctuation characters as defined in Assignment #1. You must have at least one (default) tokenization strategy.

Classes `indexer` and `query_result`: The `indexer` is responsible for storing and maintaining your document index. An object `idx` of class `indexer` holds the data structures created from the input documents. It has a default constructor, which creates an empty index. A function `size()` returns the number of documents in the index. A new document can be read into the index through an overloaded extractor (`operator>>`).[4] A function `normalize()` computes the tf-idf weights based on the number N of indexed documents. A function `query(string, int)` is used to query the index with the provided `string`. By default, it returns the top-10 results, but this can be overridden on a per-query basis (optional second argument). If the index has not been normalized, attempting to query it will throw an exception (adding a new document after normalization also results in a non-normalized index). The `query()` function returns a `vector<query_result>`, where each result object has a `document` and its `score`. Class `indexer` also provides access to its indexed documents with an overloaded `operator[]`: e.g., `idx[3]` would return the fourth document in the index, as an object of class `document`.

For all these classes, overload the inserter (`operator<<`) to provide meaningful debug output. You can add additional classes if you like, but these must not duplicate the functionality of the classes above. Note that you are responsible for coming up with an object-oriented design that makes good use of these classes, so that they collaboratively solve the two stated tasks.[5]

[3] Thus, an object `sw` of the `stopword` class works as a function object that you can call like `sw("hello")`

[4] That is, it works as `d >> idx`, where d is a `document` object and idx is an `indexer` object

[5] To brainstorm the program design within your team, write each class name on an index card, following the *Class-responsibility-collaboration card* (CRC card) methodology: https://en.wikipedia.org/wiki/Class-responsibility-collaboration_card