COMP 345 Advanced Program Design with C++

Fall 2017

Assignment #3

Due date (Moodle Submission): November 7
Due date (Lab Demo): November 8 & 15

Task 3: "Essay Generator" (9%). Develop a C++ program that can write an English essay on a given topic, provided with a set of source documents. For example, it will be possible to give your program a set of newspaper articles, and ask it to write an essay in 500 words on a topic like: *"What countries are or have been involved in land or water boundary disputes with each other over oil resources or exploration? How have disputes been resolved, or towards what kind of resolution are the countries moving? What other factors affect the disputes?"*

For this task, your program has to be able to:

1. Index a set of text files, provided in the same form as in Assignments #1 & #2;
2. Read an essay topic (like in the example above) from another text file;
3. Generate an essay with a specific length (e.g., 500 words) using the indexed documents as source material.

Some example documents, with corresponding query topics, are available for download on Moodle. You must include your program's output for these in your assignment submission.

Document indexing. Instead of building a document-term matrix like for Assignments #1 & #2, you now have to create a *sentence-term* matrix. For this, you will have to add code that detects the end of a sentence in a document, for example, by looking for ".", "?", or "!" followed by a space or newline. In other words, you now treat each sentence like a "mini-document" containing only this single sentence.

Processing the query. Rather than taking a query from a user, like in Assignment #2, your query now is the topic that you read from the provided question file. This topic is then converted into a query vector $\vec{q}$ like before.

Essay generation. You generate the essay by querying your (normalized!) index with the topic query vector $\vec{q}$, using the same similarity function as described in Assignment #2. As a result, you will get a *ranked list of sentences*. You can now generate the essay for the question based on the top-ranked sentences, such that the length of all result sentences combined is shorter or equal to the requested length (e.g., ≤ 500 words). For the output (essay) this ranked list of sentences is then sorted: first by document, and then by its original position in the document, so that sentences appear in their original order.[1]

Congratulations, you now have your very own *automatic summarization* system. It is generating a special kind of summary, called a *focused summary*, which answers a specific question of a user, given a set of documents.[2] And if you can make your program smarter, maybe someone will even buy it...[3]

[1] That is, if s_1 appeared before s_2 in a document, it is printed before s_2 in the summary, even if s_2 has a higher rank than s_1.

[2] There have been international competitions on developing systems solving this problem, see for example the DUC/TAC competitions organized by the U.S. National Institute of Standards and Technology (NIST), which were partially supported by the U.S. Department of Defense (DoD), see e.g. http://duc.nist.gov and http://www.nist.gov/tac/.

[3] In 2013, Yahoo! bought the "Summly" summarization app developed by 17-year-old Nick D'Aloisio for US $30 million.

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; use `friend` functions where appropriate to access private members; access `private` members in derived classes through `protected` functions, and make proper use of inheritance (e.g., use `virtual` functions for polymorphism and do not override non-virtual functions in `public`ly derived classes).

d) Write three separate `main` programs, using the *same* classes (see *e)* below): a new `summarizer.cpp` for Task 3, as well as updated versions of `indexing.cpp` *(renamed)* that implements Task 1 from Assignment 1 and `googler.cpp` for the search Task 2 (you will have to demo all three main programs).

e) Design your new code around the following classes and methods (classes that are not mentioned work as defined for Assignment #2):

Class `index_item`**:** Introduce a new abstract base class (ABC) `index_item` that has two subclasses: `document` (works as defined for Assignment #2) and `sentence`. Class `sentence` has an additional `private` field `pos`, which is the start position of the sentence (character offset) within its document. Provide an accessor function `getPos`. It also overrides `size` to return the number of words in the sentence. Move code that is common to both subclasses into the base class.

Class `abstract_tokenizer`**:** Introduce a new abstract base class `abstract_tokenizer` with two subclasses: `word_tokenizer` and `sentence_tokenizer`. The `word_tokenizer` works like the tokenizer in Assignment #2, splitting the input into words. The `sentence_tokenizer` splits its input text into sentences. Pay attention to abbreviations in texts: for example, *"Dr. Witte teaches COMP 345"* must not be split into two sentences!

Class `indexer` also becomes an abstract base class. Subclasses are `document_indexer` and `sentence_indexer`. The `document_indexer` works like defined in Assignment #2, indexing and querying complete documents. The `sentence_indexer` splits documents into sentences while building the index and queries the resulting sentence-term matrix. Change the `operator[]` to return an `*index_item`, i.e., a pointer to an `index_item`. Override the `query` function for the `sentence_indexer`, so that the second `int` argument now defines the maximum total words in the returned sentences (e.g., when you set it to 500 words, it will return as many top-ranked sentences as possible to fit in the total length of 500 words).

Class `query_result`**:** A query_result is now a tuple of (`*index_item`, `score`).

You code must make use of polymorphism, where appropriate. 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. As for Assignment #2, you are responsible for coming up with an object-oriented design that makes good use of these classes, so that they collaboratively solve the stated tasks (e.g., using the mentioned CRC method).