CECS 429 - Project Milestone 1

Milestone demo due October 3.

Overview

In this project milestone, you will piece together and extend many of the search engine features you have already programmed in this class. You may work in teams of up to 3 people, but as you will see, more people in a group will necessitate additional work.

You will program an application to index a directory of files using a positional inverted index. You will ask the user for the name of the directory to index, and then load all the documents in that directory and construct your index with their contents. You will then process the user's Boolean search queries on the index you constructed.

This first milestone will use a text-based interface as in your homework assignments. When processing a query, you must display the name of each document that matches the request, and then allow the user to (optionally) select a document from the result list. If the user selects a document, its **original** text (not processed, stemmed, etc.) should be displayed.

Requirements

These **mandatory** requirements must be completed by all groups.

Corpus:

The official corpus for this project will be the full `all-nps-sites.json` corpus, split into individual document files. However, there will be a future need to support multiple corpora in the search engine, so your program will begin by asking the user to enter a folder to index. You will then build your index with the documents in that folder.

Building the index:

Indexes: You will maintain one index for your corpus: the `PositionalInvertedIndex`, a positional index as discussed in class, where postings lists consist of <documentID, [position1, position2, ...]> pairs. Use `NaiveInvertedIndex` as a reference point. You may need to write your own simple `PositionalPosting` class to pair an integer document ID with a list of integer document positions; your index can then map a string term to a list of `PositionalPosting` objects. The `addTerm` method will need an additional parameter for the term's position.

The index should consist of a hash map from string keys (the terms in the vocabulary) to **(array) lists of postings**. You must not use any other hash maps or any **"set"** data structures in your code, only **lists**.

Document processing: Using the framework already built for `SimpleTokenStream` (or writing your own), process each document. For each token read from the document, perform these steps to normalize the token to a term:

1. Remove all non-alphanumeric characters from the **beginning and end**.
2. Remove all apostropes (single quotes).
3. For hyphens in words, **do both**:
   (a) Remove the hyphens from the token and then proceed with the modified token;
   (b) Split the original hyphenated token into two tokens without a hyphen, and proceed with both. (So the token "Hewlett-Packard" would turn into `HewlettPackard`, `Hewlett`, and `Packard`.)
4. Convert the token to lowercase.
5. Stem the token using an implementation of the Porter2 stemmer. Please do not code this yourself; find an implementation with a permissible license and integrate it with your solution.

Then index the document using the term and its position in the file (0-based).

Query language:

We will support user queries in the form

$$Q_1 + Q_2 + \cdots + Q_k$$

where the + represents "OR", and we call each Q_i a **subquery**, defined as a sequence of **query literals** separated by white spaces. A **query literal** is one of the following:

1. a single token; or
2. a sequence of tokens that are within double quotes, representing a phrase literal

Each of these two cases are called **positive literals**, because they must be present in a document if that document is to satisfy the corresponding Q_i. (More on positive and negative literals below.) Note that every Q_i has at least one positive literal. Examples:

- `shakes "Jamba Juice"`
  here, there is one subquery Q_1 with two positive literals (`shakes` and `"Jamba Juice"`). We want all documents with `shakes` and with the phrase `"Jamba Juice"`.
- `shakes + smoothies mango`
  here, we have two subqueries, Q_1 =`shakes` and Q_2 =`smoothies mango`. We want all documents with `shakes` or both `smoothies` and `mango`.

After parsing a query, your index will process one Q_i at a time, merging the postings lists for each query literal in the sequence. After each Q_i has been processed, the final result set will be combined with "OR" logic and shown to the user. In the query `"Jamba Juice" shakes + mango smoothies`, you would first merge the results from `"Jamba Juice"` and `shakes`; then merge `mango` and `smoothies`; then OR the final result sets.

You may assume that the user always types in a properly-formatted query, and do not need to do error checking or reporting. Note that this query language does not support the `NOT` operator. It also does not support queries of the form `smoothies (mango + banana)`. Finally, note that each token in the query should be processed in the same way that the documents are.

Special queries: Your engine must also handle a few "special" queries that do not represent information needs, but instead are used for administrative needs. If the user query starts with any of these strings, then perform the specified operation **instead** of doing a postings retrieval.

- `:q` – exit the program.
- `:stem` *`token`* – take the token string and stem it, then print the stemmed term.
- `:index` *`directoryname`* – index the folder specified by *`directoryname`* and then begin querying it, effectively restarting the program.
- `:vocab` – print all terms in the vocabulary of the corpus, one term per line. Then print the count of the total number of vocabulary terms.

Program flow:

Your program should follow this general flow of operation:

1. At startup, ask the user for the name of a directory that they would like to index.
2. Index all documents in the directory using the positional index as outlined above.
3. Loop:
   (a) Ask for a search query.
      i. If it is a special query, perform that action.
      ii. If it is not, then parse the query and merge the postings lists as necessary.
         A. Output the names of the documents returned from the query, one per line.

B. Output the **number of documents** returned from the query, **after** the document names.

C. Ask the user if they would like to select a document to view. If the user selects a document to view, print the entire text of the document to the screen.

Additional Requirements

You must select and implement N additional features from the following list, where N is equal to...

- the number of people in your group...
- plus 1, if at least one member of your group is enrolled in CECS 529.

Finally, you **must** select at least one Category A option.

Category A:

Your own corpus: Choose a corpus of documents to use with your search engine. Download (or otherwise retrieve) your chosen corpus, which must contain at least 10000 documents totaling at least 10 MB of uncompressed disk space.

You must submit a **corpus proposal** to me by September 15 in order to select this feature for your project. Your proposal must be typed, and must contain: a description of the corpus you would like to use; *why* you would like to use this corpus; how you will retrieve the corpus; any special processing rules needed when tokenizing the corpus.

I probably will not accept a corpus proposal if you simply plan to download some novel and index its chapters like we have done with *Moby Dick*. Your corpus will need to require a good amount of effort to retreive/process, and represent some specific domain that is of interest to you.

NOT queries: Amend your query language to support the boolean NOT operator as a minus sign ("`-`"). To support this feature, edit the definition of **query literal** above to:

1. a single token
2. a sequence of tokens that are within double quotes, representing a phrase query
3. **a query literal preceeded by the negation symbol -**

We say that a literal with a `-` is a **negative literal**, and *require* that each Q_i in the query have at least one **positive** literal. Knowing this, you should carefully select the order of terms when processing a query literal: process and merge all positive literals, then merge with any negative literals. **Note:** your NOT operator must also work with phrase queries. Your language now supports queries like `-"Jamba Juice" shakes smoothies` and `shakes + smoothies -mango`.

Expanded query parser: Our base Boolean query language only supports disjunctive normal form (DNF) queries: all queries are "ORs of ANDs", being one or more AND queries joined with ORs.

Tweak your query parser and execution engine to redefine the term **query literal** as:

1. a single token
2. a sequence of tokens that are within double quotes, representing a phrase query
3. a pair of parentheses surrounding a DNF query.

Your language now supports queries like `smoothies (mango + banana) + "vanilla shakes"`, for someone wanting to know about either mango smoothies or banana smoothies, OR wants to know about vanilla shakes.

Wildcard queries: Amend the query parser to support wildcard queries with arbitrary numbers of * characters representing wildcards. Implement 1-, 2-, and 3-gram indexes for each vocabulary **type** in the vocabulary. Represent the k-gram index with a class `KGramIndex`, with a method to add a string type to the index (thereby generating 1-, 2-, and 3-grams for the type and inserting them into an inverted index structure),

and a method to retrieve a list of types that contain a given k-gram. When reading a wildcard token in a user query, generate all k-grams for the wildcard token, then merge the postings list of each k-gram. The merged list contains all candidate type strings; implement a post-filtering step to ensure that each candidate matches the wildcard format. OR together the postings for the processed/stemmed term from each final wildcard candidate and then merge those postings as needed in the rest of the query.

Soundex algorithm: Read the textbook chapter 3.4 regarding phonetic correction using the "soundex algorithm," then implement the algorithm so your search engine's users can search boolean queries on the **authors** of documents in your corpus, using tolerant retrieval for misspelled names via the soundex algorithm.

The National Park Service corpus does not have author information in its documents. To complete this option, you will need to download a a different corpus: `mlb-articles-4000.zip` from BeachBoard, containing articles about Major League Baseball. The JSON format for these articles contains an `author` field.

Create a class to represent a soundex index. A soundex index maps from soundex hash keys (four-letter strings) to document IDs that contained a string which produced the hash. When indexing a document, after you have processed all the tokens in the body of the document, process the `author` of the document as well, by hashing each term in the author field (the author's first and last names, presumably) using soundex, and inserting a posting into the soundex index.

When querying a corpus, allow the user to search the author soundex index by specifying a query of `:author _______`. A single term may follow `:author`, and that is the term you should hash and retrieve postings for. List all documents that matched the author query.

Foreign language indexing: Once again, I am an ignorant American, so "foreign" here means non-English. Expand your search engine to handle non-English language documents and queries. The language you choose must have different indexing/parsing rules than the simple rules we use for English language, and you must program your engine to handle those rules. Example: you may choose German if you design a software module to split compound nouns, so that *Computerprogrammierer* is indexed separately as *computer* and *programmierer*. (You will probably have to do your own research into how this is done.) Your engine must still operate on English documents so I can test it, and you may use two different "modes" for operating on English or non-English collections if desired.

You must submit a summary of your language of choice, the issues that make it harder than indexing English, and a summary of how you will solve those issues when you turn in your project proposal.

Note that you will need to also complete Category A option "Your own corpus" for this option to really make any sense.

Your own interests: Are you interested in something else that we've talked about in lecture? Maybe you want to do some research and implementation on your own? Come chat with me and let me know.

Category B:

NEAR operator : Implement the NEAR/K operator in Boolean retrieval mode. Given a value k, do a positional merge between the terms to the left and right of the NEAR operator, selecting documents where the second term appears at most k positions away from the first term. (We will say that the order of the terms matters: "angels love baseball" will match the query "angels NEAR/2 baseball" but not "baseball NEAR/2 angels".) The three components of a NEAR operator (the first token, the "NEAR/K", and the second token) should be considered a class of **query literal** in our Boolean query language, and can appear in any subquery in a query, and the results of a NEAR may need to be merged with other literals. However, NEAR will never appear in a phrase, and you are *not* required to perform "NOT NEAR" queries if you are implementing the NOT operator.

Biword index: When indexing the corpus, additionally build a biword index. When the user enters a phrase query literal, use the biword index to satisfy the query if and only if the phrase query contains only two terms; use the positional index otherwise. The biword index can be a nonpositional index, that is, you will record document IDs for each biword entry, but not the position of the biword in the document.

Graphical user interface: Instead of a text interface, program a graphical user interface. When the program starts, show a dialog that lets the user select a directory to index. When the index is complete, show a text field for the user to type their queries. Upon submitting the query, show a list of document titles that match the query. When the user double-clicks a result in the list, open a new window showing the contents of the document.

You must provide GUI-appropriate alternatives to the "special queries" mentioned in the required features.

Unit testing framework: Implement a unit testing framework for portions of your search engine. Using a style and method appropriate to your language of choice, design, implement, and run unit tests to verify the correctness of the following portions of your search engine:

1. **Positional inverted index**: write a set of 5 simple text documents with a combined vocabulary of 10-15 words. Build a positional inverted index **by hand**. Write unit tests to constuct an index over your documents, then verify that the positional postings match the expected output based on your hand-built index. You do not need to verify each word in the vocabulary, but should be sure to test words that occur more than once in a particular document.
2. **Query processing**: using your example corpus from above, write unit tests to verify the correctness of your search engine results. Write tests for each kind of query you have to implement: phrase queries, AND queries, OR queries, and NOT queries (if you chose this option). Verify the results match your expected outputs by solving the queries by hand. Make sure you write queries for which there is no result set, and queries for which all documents are relevant.
3. **Other modules**: you must also write tests for the other optional modules you have selected for your project, appropriate to each module.

Summary

I realize this is a long document and it may be hard to keep track of what's required. Here is a brief list of important things to keep in mind. (This is NOT exhaustive, you are ultimately responsible for everything in this document even if it isn't listed in this summary.)

You must:

1. Build a positional inverted index over a user-specified directory of text files.
2. Process tokens from files with Porter stemming and special rules for other characters; and use the same rules for user queries.
3. Read Boolean queries from the user and merge postings lists for the query (both AND and OR merges).
4. Support phrase queries with quotation marks via the positional index.

You must choose one of these options for each person in your group. If any group member is enrolled in 529, select one additional option. At least one option **must** come from Category A.

1. Category A:
   (a) Your own corpus (must submit proposal to instructor)
   (b) NOT queries
   (c) Expanded query parser
   (d) k-gram index for wildcard queries
   (e) Soundex index
   (f) Non-English indexing (requires proposal)

   (g) Your own research interest

2. Category B:
   (a) NEAR/K operator
   (b) Biword index
   (c) GUI
   (d) Unit tests