CS6200 Information Retrieval
Fall 2019
Instructor: Omar Alonso

Homework #2. Build your own indexing and query a TREC data set

Objective: process a collection of documents, create an inverted index on titles, and run queries.

This programming assignment involves the following components:

1. A tokenizer (10 points)
2. An indexer (40 points)
3. A query processor which runs queries from an input file against your index (35 points).
4. Provide a table with text statistics (5 points).
5. Chart Zipf and Heaps laws (10 points).

Prerequisites

1. Use the AP89_DATA.zip collection.

Document Indexing

Create an index of the downloaded corpus. The documents are found within the ap89_collection folder in the data .zip file. You will need to write a program to parse the documents and index the title field (<HEAD></HEAD>).

The corpus files are in a standard format used by TREC. Each file contains multiple documents. The format is similar to XML, but standard XML and HTML parsers will not work correctly. Instead, read the file one line at a time with the following rules:

1. Each document begins with a line containing <DOC> and ends with a line containing </DOC>.
2. The first several lines of a document's record contain various metadata. You should read the <DOCNO> field and use it as the ID of the document.
3. The tags <HEAD> and </HEAD> describe the title of the document.
4. The document contents are between lines containing <TEXT> and </TEXT>.
5. All other file contents can be ignored.

Tokenizing

The first step of indexing is tokenizing documents from the collection. That is, given a raw document you need to produce a sequence of tokens. For the purposes of this assignment, a token is a contiguous sequence of characters which matches a regular expression (of your choice) – that is, any number of letters and numbers, possibly separated by single periods in the middle. All alphabetic characters should be converted to lowercase during tokenization, so bob and Bob and BOB are all tokenized into bob. See class notes for more examples.

You should assign a unique integer ID to each term and document in the collection. For instance, you might want to use a token's hash code as its ID. However, if you decide to assign IDs, you will need to be able to convert tokens into term IDs and covert doc IDs into document names in order to run queries. This will likely require you to store the maps from term to term_id and from document to doc_id in your inverted index. One way to think about the tokenization process is as a conversion from a document to a sequence of (term_id, doc_id, position) tuples which need to be stored in your inverted index.

For instance, given a document with doc_id 20: “The car was in the car wash.”, the tokenizer might produce the tuples: (1, 20, 1), (2, 20, 2), (3, 20, 3), (4, 20, 4), (1, 20, 5), (2, 20, 6), (5, 20, 7)

with the term ID map:

1: the
2: car
3: was
4: in
5: wash

Indexing
The next step is to design and implement an inverted index. Your task is to design such indexing scheme. The inverted list for a term must contain the following information: a list of IDs of the documents which contain the term, along with the TF of the term within that document and a list of positions within the document where the term occurs. (The first term in a document has position 1, the second term has position 2, etc.). As part of index creation, you can decide (or not) to use stop words and/or stemming. Please describe your design choices.

Hint: try your indexing code with a small set of documents first and then scale to the rest of the collection.

Query Execution
Write a program to run the queries in the file `hw1_test_queries.txt`. You should run all queries and output the top-k results (document ID and title) for each query to an output file. You decide the value for k.

Text statistics
Write a program to run produce the following statistics based on the index that you created:

- Total number of documents
- Total word occurrences
- Vocabulary size
- Number of words occurring > 1000 times
- Number of words occurring once

Chart Zipf and Heaps laws
Produce two different figures for Zipf and Heaps.

What you need to submit

1. Java source code for items 1-4
2. A description of your indexing design. How are you approaching the problem, your specific design choices, and other information that you think is important to describe.
3. Two figures that represent Zipf and Heaps.
4. Documentation on how to compile and run the code. I’ll be testing your work.