COL 864 Assignment 1 - Boolean and Vector-space Models

3rd August 2018 - Version 1.0

Deadlines

1. First submission: 13th August 2018 11:59 PM IST
2. Second and final submission: 19th August 2018 11:59 PM IST

Weightage

Assignment is evaluated against 100 marks. In the final scheme of things, it will be scaled to 1/5th total assignment weightage.

Instructions

1. This programming assignment is to be done by each student individually. Do not collaborate -either by sharing code, algorithm, and any other pertinent details- with each other.
2. All programs have to be written either using Java or C/C++ programming languages only.
3. A single tar/zip of the source code has to be submitted. The zip/tar file should be structured such that
   - upon deflating all submission files should be under a directory with the student's registration number. E.g., if a student's registration number is `20XXCSXX999` then the tarball/zip submission should be named 20xxcsxx999.{tgz|zip} and upon deflating **all contained files** should be under a directory named ./20xxcsxx999 only. If this is not followed, then your submission will be rejected and will not be evaluated.
   - apart from source files the submission tarball/zip file should contain a build file (allowed build systems are Maven and Ant for Java, Makefile for C/C++). It is the responsibility of each student to ensure that it compiles and generates the necessary executables as specified. **Note that we will use only Ubuntu Linux machines to build and run your assignments.** So take care that your file names, paths, argument handling etc. are compatible.
4. You *should not* submit data files and NLTK libraries. If you are planning to use any other "special" library, please talk to the instructor first (or post on Piazza).
5. Note that your submission may be evaluated using different collections. Only assumption you are allowed to make is that they will all be in English, are plain-text (e.g., not HTML), have sentences terminated by a period, each document is a separate file, and all documents are in one single directory. The collection directory will be given as an input (see below).
6. **Note that there are two submission deadlines – what needs to be submitted in each deadline is given at the end of the document.**

1 Assignment Description

We studied Boolean and Vector-space retrieval model in the course. In this assignment you will put the concepts we studied into practice by building an end-to-end retrieval system for English, perform a toy-scale performance evaluation of Boolean and Vector-space models.

You should use NLTK (Python library downloadable from http://nltk.org) for some of the required preprocessing of documents as required – see below. **You need not submit NLTK libraries, but must submit your wrapper programs –written so as to invoke Python NLTK library functions from Java/C/C++, if any**.

1.1 Expected Features of the System

You are going to build a retrieval system which supports Extended Boolean as well as Vector-space retrieval over a large English document collection. Specifically, it should have the following user-level features:

1. **Boolean retrieval:** Your system should support both conjunctive (specified by `&`) and disjunctive (specified by `|`) multi-word queries (*ignore negation*). Operator grouping has to be supported using the paranthesis. That is, a query like "`((Disco & Dancer) | (Mithun Chakravarti)) & (Bappi & Lahiri)`" should be parsed and interpreted correctly.

   - **It is also expected that you support prefix queries.** E.g., a query-term such as "`mit*`" should match all terms which start with "mit" like `mithun`, `mitt`, `mitra`, and so on.

2. **Extended Boolean Retrieval:** Prefixing a Boolean query term with '`N:`' should restrict that query term to match only the *named entities* in the document – you can use NLTK or Stanford CoreNLP to get the named entities. As an example, query "`N:new* & mayor`" should match a document that contains "New York's mayor ..." but not a document containing only the occurrence of "newly elected mayor".

3. **Zone-weighted Scoring:** Documents have to be ranked using a simple zone-weighted scheme that assigns high score to matching paragraphs in their order in the document. In other words, let $I(ij)$ be an indicator function that takes value 1 iff a paragraph p_{ij} in document d_j matches the specified query (not just a term or a part of the query), 0 otherwise. Then, the total weight of the document, $W(d_j)$ is given by,

$$W(d_j) = \sum_{\forall p_{ij} \in d_j} I(ij)\ \frac{1}{2^i}.$$

   Note: If the query is not matched within a paragraph, then the document does not get any weight as per the definition, but it still has to be retrieved if the document matches query. As an example, consider the following scenario:

Doc 1	"Machiavelli was born on the 3rd of May 1469. The period of his life almost exactly coincides with that of Cardinal Wolsey. The functions of his Council were extremely varied, and in the hands of their Secretary became yet more diversified. They represented in some sense the Ministry for Home, Military, and especially for Foreign Affairs."
Doc 2	"What manner of man was Machiavelli at home and in the market-place? It is hard to say."
Query	`machiavelli & home`

   Then, it expected that **both Doc 1 and Doc 2** will match the query and thus will be retrieved. However, the Doc 2 will have a score of 0.5 since the first (and the only) paragraph in Doc 2 matched the query entirely, but Doc 1 will have a weight of 0.

4. **Extended Vector-space Retrieval:** Standard vector-space retrieval using the $tf_{i,j} = (1 + \log f_{i,j})$, $idf_i = \log(1 + \frac{N}{df_i})$ where $f_{i,j}$ is the count of the term i in document j, N is the size of the document collection, df_i is the count of documents containing term i. Similar to Boolean retrieval, it should also support (a) prefix search, (b) restriction to named entities, and (c) zone-based weighting on top of the standard tf-idf score. Note that since vector-space model by default supports only disjunction, you can ignore Boolean operators in the query (if any).

1.2 Program Structure

In order to achieve these, you are required to write the following programs:

1. **Inverted indexing of the collection:** Program should be named as `invidx_cons.{c|cpp|C|h|hpp|java}`. After building, its synopsis is:
   `invidx [--stemming={0|1}] [--stopwordremoval={0|1}] coll-path indexfile`
   where `stemming=1` enables stemming, `stopwordremoval=1` removes stopwords from the index, `coll-path` specifies the directory containing the document collection files (each document is a separate file), and `indexfile` is the name of the generated index files. The program should generate two files: `indexfile.dict` contains the dictionary and `indexfile.idx` contains the inverted index postings. You can use stemmer and English stopword remover from NLTK package.

   dictionary should be a human readable file with each line containing the following structure –
   `<indexterm>:<df>:<offset-to-its-postingslist-in-idx-file>`

   postings list file can be a binary file containing the list of document identifiers (and other statistics) for each index term. Note that it could also have additional information such as reverse mapping of document identifiers to document filenames etc. to help in result generation. There is no restriction on how these (and any other) info will be stored in the postings list file.

   You can structure your source-code into multiple files if necessary, but all of them should be compiled through appropriately named main source file and should generate a single executable file / classfile with the specified name). Also note that your program should not assume that the collection is preprocessed using NLTK (i.e., you should invoke required NLTK methods within your program).
   Construct an appropriate inverted index consisting of postings-list and dictionary structure which are stored on disk in the specified filenames.

2. **Search and rank:** There should be two programs `boolsearch` and `vecsearch` which do extended Boolean and vector-space retrieval respectively. Both of these two programs take the following arguments:

`--query queryfile`	a file containing queries, with each line corresponding to a query (Boolean or otherwise)
`--cutoff k`	the number k (default 10) which specifies how many top-scoring results have to be returned for each query
`--output resultfile`	the output file named `resultfile` which is generated by your program, which contains the filenames of all documents that have top-k (k as specified) highest-scores (i.e., filenames from the collection, not just an document identifier, and all documents whose score is in the top-k scores) and their scores in each line. Results for each query have to be separated by **2 newlines**.
`--index indexfile`	the index file generated by `invidx_cons` program above
`--dict dictfile`	the dictionary file generated by the `invidx_cons` program above

You will be supplied with a test document collection on 10th August 2018.

1.3 Submission Plan

All your submissions should strictly adhere to the formatting requirements given above.

- First submission on 13th August - you should have a fully working implementation of extended Boolean retrieval but **not necessarily including** prefix search, zone-weighting and top-k cutoff
- Second submission on 19th August - you should have a fully working implementation of extended Boolean and vector-space retrieval.

1.4 Tentative breakup of marks assignment

In general, a submission qualifies for evaluation if and only if it adheres to the specifications given above (arguments, structure, use of external libraries, correct output format, input format adherence, etc.). Given this requirement, the marks assignment for correct implementation of:

inverted index (postings list + dictionary)	20
basic Boolean retrieval (including all Boolean expressions)	10
prefix search	15
paragraph restriction and zone weighting	15
named entity restriction	5
top-k cutoff	5
basic vector-space retrieval	15
extended vector-space retrieval	15
Total	100