COL 764 Assignment 1 - Vector-space Retrieval

2020 - Version 1.0

Deadlines

1. Deadline for final submission of the complete implementation, and the report on the algorithms as well as performance tuning: October 21st 2020, 11:59 PM
2. The form for submitting self-reported metrics on the training data will be opened from October 14th, 2020.

Weightage

This assignment is evaluated against 100 marks. The tentative breakup of marks is given in the end of the document.

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 Python/Java/C/C++ programming languages only. Anything else requires explicit prior permission from the instructor.
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 mechansim if needed (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. If you are planning to use any other "special" library, please talk to the instructor first (or post on Teams).
5. Note that your submission will be evaluated using larger collection. Only assumption you are allowed to make is that all documents will be in English, have sentences terminated by a period, each document is a separate file, and all documents are in one single directory. The overall format of documents will be same as given in your training. The collection directory will be given as an input (see below).
6. **Note that there will be no deadline extensions. Apart from the usual "please start early" advise, I must warn you that this assignment requires significant amount of implementation effort, as well as some 'manual' tuning of parameters to get good speed up and performance. Do not wait till the end.**

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 Vector-space retrieval models. You are expected to implement an *efficient* and *effective* retrieval system that uses a slightly enriched textual data for its input.

Specifically, your program should have the following features:

1. **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.

2. **Prefix search:** A query term such as "`mit*`" should match all terms which start with "mit" like `mithun`, `mitt`, `mitra`, and so on. The score will be computed as the sum of individual $tfi, j \cdot idf_i$ scores of the matching terms (i.e., as a disjunction of all matching terms to the prefix).

3. **Restriction to named entities:** The document collection has been preprocessed using NLTK/StanfordNLP toolkits to tag terms as belonging to `PERSON`, `ORGANIZATION`, `LOCATION` classes (read the attached FAQ document). Prefixing a query term with '`N:`' should restrict that query term to match only the *named entities* in the document.

   As an example, query "`N:new*`" should match a document that contains "<LOCATION>New York< /LOCATION>'s mayor ..." but not a document containing only the occurrence of "newly elected mayor".

   We will use `P:` for restricting to only `PERSON`s, `O:` for `ORGANIZATION`s, `L:` for `LOCATION`s, and `N:` for any of the three (i.e., any tagged noun).

   The entity tagging is not perfect. As a result there are many noun-phrases which are not tagged during preprocessing. You are not expected to run any further fixes for this.

1.1 Task

You will be given a small-sized benchmark document collection from TREC, which has been preprocessed by NLTK/StanfordNLP. At the same time, you are also given a portion of the TREC topics (i.e., queries) and their judgements (i.e., qrels) on these documents. The task is:

1. Build an *efficient* inverted indexing system that works on at least 2-times the document collection released. Efficiency is measured by

   (a) Wall-clock time taken to completely index all the documents, and write the vocabulary and index files to the disk.

   (b) Size of the vocabulary and index files – smaller the better. It is important that you do not simply object-dump an in-memory datastructure, but write them out as a binary file (run-length encoding, or variable byte encodings could be tried to further reduce the size).

   Correctly implemented top-5% most compact indexes will receive additional 10 points in the assignment (note that **all** other features should be implemented).

2. Build an effective retrieval system that takes keyword queries as inputs, and using appropriate tuning techniques achieves a high retrieval quality using vector-space retrieval. Retrieval quality, is measured using nDCG and F_1 scores. Top-5% of the leader-board in retrieval quality in either nDCG or F_1 will receive additional 10 points in the assignment. Note that retrieval quality can be improved by methods such as:

(a) Preprocessing the queries to identify named entities (using NLTK/StanfordNER or any other technique), and using them to further improve the quality of retrieval. You should follow the same convention as outlined above for denoting the named entity restrictions in the query.

(b) Using query expansion techniques (which we will study soon), to further refine the query to achieve improvements in precision and recall.

(Note that the above two are only two suggested methods. You can consider other approaches)

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.{py|c|cpp|C|java}`. After building, its synopsis is:

   ```
   invidx coll-path indexfile
   ```

   where,

`coll-path`	specifies the directory containing the files containing documents of the collection, and
`indexfile`	is the name of the generated index files.

   The program should generate two files:

   (a) `indexfile.dict` contains the dictionary and

   (b) `indexfile.idx` contains the inverted index postings.

   You can use stemmer and English stopword remover from NLTK package if needed, but take care not to reduce the retrieval quality as a result.
   postings list file can be a binary file containing the list of document identifiers (and other statistics) for each index term. There is no restriction on how these (and any other) info will be stored in the postings list file.

   You should also implement a helper tool for printing the dictionary, whose synopsis is:

   ```
   printdict indexfile.dict,
   ```

   which prints the dictionary out in a human-readable form to the screen, with each line containing the following structure –

   ```
   <indexterm>:<df>:<offset-to-its-postingslist-in-idx-file>
   ```

   where offset in index file is the number of bytes from the start of the file where the corresponding postings list starts.

   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 names specified above.
   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 another program called `vecsearch` (in corresponding source file `vecsearch.{py|c|cpp|C|java}` for performing vector-space retrieval. It takes the following four

command-line arguments:

`--query queryfile`	a file containing keyword queries, with each line corresponding to a query
`--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 document ids of all documents that have top-k (k as specified) highest-scores and their scores in each line (note that the output could contain more than k documents). 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

Result Output Format: We will make use of trec_eval tool for generating nDCG and F_1 scores. Therefore, it is **very** important that you follow the exact format that is required by the tool. The trec_eval tool is freely downloadable from `https://github.com/usnistgov/trec_eval`. The result file format instructions are as follows:

Lines of results_file are of the form
```
030 Q0 ZF08-175-870 0 4238 prise1
qid iter docno rank sim run_id
```
giving TREC document numbers (a string) retrieved by query qid (a string) with similarity sim (a float). The other fields are ignored, with the exception that the run_id field of the last line is kept and output. In particular, note that the rank field is ignored here; internally ranks are assigned by sorting by the sim field with ties broken deterministicly (using docno). Sim is assumed to be higher for the docs to be retrieved first. File may contain no NULL characters. Lines may contain fields after the run_id; they are ignored.

[taken from the comments in `format.c` file in the trec_eval repository].

You will be supplied with a test document collection, queries and qrels not later than 10th October 2020.

1.3 Submission Plan

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

- Submission of your source code on 21st October, and the final version of results.
- You should also submit a README for running your code, and a PDF document containing the implementation details as well as any tuning you may have done.

Leaderboard: We will have a form-based submission of index construction time, size, nDCG and F_1 scores starting from 14th October 2020. You can submit it as many times as you want (a subset of these metrics as well), and the leaderboard of submissions will be continuously updated and shared to all.

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
vector-space retrieval	15
named entity restriction	10
prefix search	10
top-k cutoff	5
buildfile	5
README and algorithmic details documentation	15
Leaderboard gains	20
Total	100